SHARP

SHARP
COMPUTER
SOFTWARE

X68000用

C COMPILER ver2.0 PRO-68K

Cユーザーズマニュアル

X68000用

C COMPILER PRO-68K ver2.0

Cユーザーズマニュアル

SHARP

はじめに

このたびは、「C compiler PRO-68K ver 2.0」をお買い上げいただき、まことにありがとうございました。
「C compiler PRO-68K ver 2.0」(以下、XC コンパイラと表記します)は、X68000 のために作られた、総合開発ツールです。本ソフトウェアを用いてプログラム開発を行う場合、メインメモリが 2MB 必要です。
ご使用に際しては、必ず本説明書に記載されている操作方法・注意事項をよくお読みいただき、正しい操作によって有効に活用されるようお願い致します。

商品構成は、下記の通りです。

XC システムディスク 1	1 枚
XC システムディスク 2	1 枚
XC ライブラリディスク	1 枚
C ユーザーズマニュアル	1 冊
C リファレンスマニュアル	1 冊
C ライブラリマニュアル VOL.1	1 冊
C ライブラリマニュアル VOL.2	1 冊
ソースコードデバッガマニュアル	1 冊
アセンブラマニュアル	1 冊
プログラマーズマニュアル	1 冊
登録カード	1 枚

<ご注意>

1. 本書の内容については万全を期して作成致しましたが、万一ご不審な点や誤り記載もれなど、お気付きのことがありましたら、もよりのシャープお客様ご相談窓口あるいはお買い求めの販売店にご連絡ください。
2. 運用した結果の影響については、1項にかかわらず責任をおいかねますのでご了承ください。
3. 本書の内容に関しては、将来予告なしに変更することがあります。
4. 本書の内容の一部または全部を無断転載することは、禁止されています。
5. 付属の登録カードは必ず弊社までご返送ください。
   無登録の方は、一切のユーザーサポートが受けられませんので、ご注意ください。
6. 本ソフトウェアを用いてソフト開発を行った商品を販売する場合の使用料(ロイアリティ)は無償ですが、マニュアルなどに本ソフトウェアを使用したことを明記ください。

「C compiler PRO-68K ver 2.0」の各マニュアルは次の内容となっています。

● Cユーザーズマニュアル

Cコンパイラ（XC）、BASIC-Cコンバータ（XBAStoC）を使ったソフト開発に必要な操作手順について説明しています。

● Cリファレンスマニュアル

Cコンパイラ（XC）の言語仕様について説明しています。

● Cライブラリマニュアル VOL.1

Cコンパイラ（XC）で利用できる豊富なCライブラリの概要と使用方法、標準ライブラリ／BASICライブラリ／日本語ライブラリ（小文字関数）のリファレンスに関して記述しています。

● Cライブラリマニュアル VOL.2

IOCSライブラリ／DOSライブラリ（大文字関数）のリファレンスに関して記述しています。

●ソースコードデバッガマニュアル

Cコンパイラ（XC）のソースレベルでデバック可能なソースコードデバッガ（SCD）の機能について説明しています。

●アセンブラマニュアル

アセンブラ（XAssembler）を使用したソフト開発に必要な操作手順、およびリンカ（XLinker）／デバッガ（XDebugger）／アーカイバ（XArchiver）／ライブラリアン（XLibrarian）／コンバータ（XConverter）の取り扱いについて説明しています。

●プログラマーズマニュアル

Human68k ver2.0上のアプリケーションソフトを開発するために便利な各種コマンドの操作方法、DOSコール、IOCSコール、デバイスドライバ、X-BASIC外部関数の作成方法について説明しています。

本書の構成

本書は、次の5章と付録から構成されています。

第1章　お使いになる前に

XCコンパイラをご使用いただく前に、本パッケージに含まれるファイルの構成や、内容について、一通りの説明を行います。
また、XCコンパイラを使ってプログラム開発を行う場合の流れについて説明します。

第2章　コンパイル

コンパイラのコマンドラインの指定方法について説明します。
XCコンパイラはスイッチの指定により、プログラム開発効率を上げることができます。
本章では、このスイッチすべてについて詳しく説明してあります。

第3章　プログラムの実行

コンパイルして完成したプログラムの実行に関して、引数のわたしかたや、プログラムがどのように引数をとり込むかを、具体的に説明してあります。

第4章　アセンブラとのインターフェイス

XCコンパイラは、C言語プログラムとアセンブラプログラムをリンクすることができます。
その際のアセンブラプログラムの作成方法を説明しています。
この章を読むことにより、Cをより深く理解することができます。

第5章　XBAStoC BASICプログラムコンバータ

BASTOCについて、コマンドの使いかたから、C言語に変換する場合の効率のよいBASICプログラムの書きかたまでを詳しく説明しています。
BASTOCを使うことにより、今まで、BASICユーザーだったかたも、すぐCユーザーになることができます。

本書の構成

第 6 章　追加 BASIC 関数

MIDI ボードを制御するために追加になった、BASIC の命令や、拡張された MML について説明しています。

付　　　録

コントロール表や、キャラクター表、またコンパイルスイッチやエラーメッセージを一覧表にして掲載してあります。

CONTENTS

第1章　お使いになる前に

第2章　コンパイル

第3章　プログラムの実行

第4章　アセンブラとのインターフェイス

第5章　XBAStoC BASICプログラムコンバータ

第6章 追加BASIC関数

付録

索引

第１章
お使いになる前に

バックアップディスクの作成
ディスクの内容
ファイルの構成
インストールと開発環境の設定
プログラム開発の流れ

本章では、XCコンパイラのインストール方法について説明します。
インストールとは、提供されたXCコンパイラのパッケージをX68000のOS、Human68kのファイルシステムに組み込むことをいいます。
もともとフロッピーディスクシステムで提供されているため、Human68kのファイルシステムとして動作しますが、プログラム開発をよりよくするため、各ハードウェアの環境整備を中心に説明します。
また、プログラム開発を行ううえで、ソースプログラム作成からコンパイルして、実行するまでの一連の流れを説明して、プログラム開発中に最低限知っていなければばならないことについてふれます。
本章では、XCコンパイラをより効果的に使いこなすうえでの予備知識として、何が必要なのかを説明しました。
なお、XCコンパイラはX68000のOSであるHuman68k上で動作しますので、Human68kのコマンドについては、ある程度の知識が必要です。
本章では、OSやコマンドについての詳しい説明は省略しました。
わからないコマンドについては、「Human68kユーザーズマニュアル」を参照してください。

1.1 バックアップディスクの作成

フロッピーディスクは長い時間使っていると、少しずつですが摩擦によって消耗します。
また、操作中にコーヒーをフロッピーディスクにこぼしてしまったなどということも、ないとはいえません。
このような場合、貴重なデータやファイルが失われることがあります。
そこで、大切なフロッピーディスクをこのような万一の事故や、誤った操作から保護するために、バックアップをする必要があります。
大切なディスクは、作業の前に必ずバックアップをとる習慣をつけてください。
また、通常の作業ではバックアップディスクを使うようにします。
オリジナルディスクは、万一に備えて大切に保管しておいてください。

1.1.1 バックアップの手順

オリジナルディスクからのバックアップ作業は、以下に示す手順により行います。
このとき、誤操作などでオリジナルディスクを破壊する危険性もありますので、慎重に行ってください。
オリジナルディスクにライトプロテクト（フロッピーディスクの切り込み部分に銀紙を貼る）をしておけば万全です。

これ以降の説明において、ドライブAはフロッピーディスクドライブの0、ドライブBはフロッピーディスクドライブの1になっているものとして説明します。
ハードディスクなどから起動した場合は、ドライブ名のA、Bが下記の説明と異なっている場合がありますので注意してください。

新しいフロッピーディスクの初期化

バックアップ用の新しいフロッピーディスクの初期化を行います。
買ってきたばかりの新しいフロッピーディスクは、初期化をしないと使用することができません。

1.1 バックアップディスクの作成

ここでは、新しいフロッピーディスクの初期化の方法を説明します。

① Human68k のシステムディスクをドライブ A にセットします。

②コマンドモードの"A>"プロンプトが表示されている状態から

```
A>format /s b:↵
```

と入力します。

③次のようなメッセージが表示されます。

```
Format version X.XX
ドライブ B：（2HD デイスク）を初期化します。
何かキーを押してください
```

バックアップ用の新しいフロッピーディスクをドライブ B にセットしてください。
ディスクをセットした後、上記のメッセージに従って、任意のキーを押します。

④新しいディスクが初期化（フォーマット）されます。

```
初期化中です……
```

⑤初期化が終了すると、終了のメッセージが表示されます。

```
初期化が終了しました
```

⑥システムファイルのコピーが行われます。

```
A：のシステムファイルをB：に転送します
```

なお、システムファイルのコピーが不要な場合は、②のコマンド入力で、／sを指定する必要はありません。
指定しない場合は、この⑥の処理は行われません。

⑦最後に次のメッセージが表示されます。

```
他のディスクを初期化しますか?＜Y／N＞
```

これ以上フォーマットを行わないならば、'N'を押します。
もしここで、続けてフォーマットを行う場合は、上記のメッセージに'Y'と答えて、③以下を繰り返します（なお、付属のオリジナルディスクは3枚ありますので、初期化するディスクも3枚必要です）。

バックアップのためのコピー

初期化されたディスクに、オリジナルディスクをすべてコピーします。
以下にコピーの手順を示します。

①"A＞"プロンプト状態を確認して、次のよう入力します。

```
A＞diskcopy　a:　b:↵
```

②次のようなメッセージが表示されます。

1.1 バックアップディスクの作成

```
Diskcopy version X.XX
ドライブＡからドライブＢへコピーします
何かキーを押してください
```

ドライブＡにオリジナルディスク、そして、ドライブＢに先ほど初期化した新しいディスクをセットしてください。
ディスクをセットした後、上記のメッセージに従って任意のキーを押します。

③オリジナルディスクからバックアップディスクにデータがコピーされます。

```
コピー中です……
```

④この後コピーが終了すると、次のメッセージが表示されます。

```
コピーを終了しました
他のディスクをコピーしますか?<Y／N>
```

オリジナルディスクは３枚ありますので、１枚目のディスクのコピーが終了した場合は'Y'と押して、②以下の作業を行います。
３枚目のコピー終了には'N'を押してください。
これでバックアップは完了します。

ドライブＡからオリジナルディスクをイジェクトして、大切に保存しておいてください。

1.2 ディスクの内容

本パッケージには、次のようなファイルが含まれています。
ディスク内のファイルがすべて揃っているかどうか確認してください。

1.2.1 XCシステムディスクNo.1

XCシステムディスクNo.1のルートディレクトリには、次のようなファイル、およびサブディレクトリがあります。

```
                ┌HUMAN. SYS
                ├CONFIG. SYS
                ├KEY. SYS
                ├USKCG. SYS
                ├BEEP. SYS
                ├STARTUP. ENV
                ├AUTOEXEC. BAT
ルートディレクトリ─┼COMMAND. X
                ├SYS(ディレクトリ)
                ├BIN(ディレクトリ)
                ├ASK(ディレクトリ)
                ├HIS(ディレクトリ)
                └ETC(ディレクトリ)
```

XCシステムディスクNo.1には、Human68kを起動するのに必要なファイルが格納されており、このディスク単体でHuman68kを起動することができます。
なお、XCコンパイラ本体（CC. X）やライブラリ、インクルードファイルなどは、XCシステムディスクNo.2に格納されています。
続いて、各サブディレクトリに格納されているファイルの説明をします。

1.2 ディスクの内容

(1) SYS

このディレクトリには、デバイスドライバ関連のシステムファイルが格納されています。

ファイル名	内　　容
PRNDRV. SYS	プリンタドライバ（シャープ・CZ 系）
PCMDRV. SYS	PCM ドライバ
ASK68K. SYS	日本語処理フロントプロセッサ
RAMDISK. SYS	RAM ディスクドライバ
SRAMDISK. SYS	SRAM ディスクドライバ
PRNDRV1. SYS	プリンタドライバ（エプソン・ESC／P 系）
PRNDRV2. SYS	プリンタドライバ（NEC・PC-PR201 系）
PRNDRV3. SYS	プリンタドライバ（NEC・NM-9300 系）
SCSIDRV. SYS	SCSI ドライバ
OPMDRV2. X	FM 音源ドライバ（MIDI 対応）
FLOAT1. X	浮動小数点演算エミュレータ（シャープ仕様）
FLOAT2. X	浮動小数点演算エミュレータ（IEEE 仕様）
HISTORY. X	ヒストリドライバ
IOCS. X	IOCS ドライバ

デバイスドライバは、CONFIG. SYS の中で

```
DEVICE = PRNDRV. SYS
DEVICE = FLOAT2. X
```

のように指定することで、システムに組み込むことができます。
なお、ファイルの拡張子が. X であるデバイスドライバは、コマンドラインから実行することで、システムに追加することもできます。

```
A>FLOAT2 ↵
```

(2)　BIN

このディレクトリには、アセンブラやリンカ、デバッカなどのユーティリティが格納されています。

コマンド名	内　　容
AS. X	アセンブラ
LK. X	リンカ
DB. X	デバッカ
SCD. X	ソースコードデバッカ
SCD. CNF	ソースコードデバッカ用コンフィグファイル
SCD. HLP	ソースコードデバッカ用ヘルプファイル
AR. X	アーカイバ
LIB. X	ライブラリアン
CV. X	コンバータ
BIND. X	バインドツール
CASE. X	ファイル名、ディレクトリ名の大／小文字変換ツール
DRIVE. X	ドライブ情報の表示、ドライブ名の変更ツール
MOVE. X	ファイル移動ツール
PROCESS. X	プロセス表示ツール
COPYALL. X	ファイル転送ツール
TERM. X	簡易ターミナルツール
TOUCH. X	タイムスタンプ変更ツール
TREE. X	ディレクトリ構造の表示ツール
WHERE. X	ファイル検索ツール
FORMAT. X	フォーマッタ
DISKCOPY. X	ディスクコピー
MAKE. X	プログラム保守ユーティリティ
PRINT. X	バックグラウンド印刷ツール
ED. X	エディタ
ED. HLP	エディタ用ヘルプファイル

(3) ASK

このディレクトリには、日本語処理フロントプロセッサ ASK68K の環境設定ファイルが格納されています。

ファイル名	内　　容
ENV1. ASK	日本語処理環境設定ファイル No.1
ENV2. ASK	日本語処理環境設定ファイル No.2
ENV3. ASK	日本語処理環境設定ファイル No.3
ENV4. ASK	日本語処理環境設定ファイル No.4
ENV5. ASK	日本語処理環境設定ファイル No.5

(4) HIS

このディレクトリには、ヒストリ機能、およびキーボードコントロールを実現するヒストリドライバが使用するファイルが格納されています。

ファイル名	内　　容
KEY. HIS	キー定義ファイル
HISTORY. HIS	ヒストリ定義ファイル
HISTORY. HLP	ヒストリのヘルプファイル

ヒストリ機能、およびキーボードコントロールに関しては、「Human68k ver 2.0 ユーザーズマニュアル」を参照してください。

(5) ETC

このディレクトリには、XC コンパイラをインストールするときに使用するバッチファイル、およびコマンドが格納されています。
ユーザーが直接使用するのはINSTALL. BATだけで、他のファイルはINSTALL. BATで使用されるファイルです。

ファイル名	内　　容
INSTALL. BAT	インストール・メインプログラム
FD. BAT	インストール・サブプログラム（FD用）
FDR. BAT	インストール・サブプログラム（FD用）
HD. BAT	インストール・サブプログラム（HD用）
YN	INSTALL. BATで使用されるテキストファイル
TOOL. X	INSTALL. BATから呼び出されるコマンド

1.2.2 XCシステムディスクNo.2

XCシステムディスクNo.2のルートディレクトリには、次のようなサブディレクトリが格納されています。

```
                ┌CC
                ├INCLUDE
                ├LIB
ルートディレクトリ┤BASIC2
                ├BC
                └SAMPLE
```

XCシステムディスクNo.2には、XCコンパイラ本体（CC. X）やライブラリファイル、インクルードファイル、BASIC関連のファイルが格納されています。

なお、このディスク単体ではHuman68kを起動することはできません。

続いて、各ディレクトリに格納されているファイルの説明をします。

(1)　CC

このディレクトリには、XCコンパイラの本体であるCC. Xだけが格納されています。

(2) INCLUDE

このディレクトリには、XC コンパイラで使用するすべてのヘッダファイルが格納されています。

なお、ERROR. H は ANSI 規格では ERRNO. H として定義されています。必要のある方は RENAME してください。

ファイル名	内　容
ASSERT. H	実行時チェックマクロの定義
AUDIO. H	ADPCM 関数の定義
BASIC. H	BASIC 組み込み関数の定義
BASIC0. H	画面処理関係などの関数の定義
CLASS. H	データ型の定義
CONIO. H	コンソール入出力関数の定義
CTYPE. H	文字チェックおよび文字変換のマクロ定義
DIRECT. H	ディレクトリ操作関数の定義
DOSLIB. H	Human68kのファンクションコールの関数の定義
ERROR. H	エラーコードのシンボル定義
FCNTL. H	低水準入出力関数の定数を定義
FCTYPE. H	CTYPE. Hで定義されているマクロの関数版の定義
FLOAT. H	算術ライブラリの定数の定義
GRAPH. H	グラフィック関数の定義
IMAGE. H	イメージ関数の定義
IO. H	ファイル操作と低水準入出力関数の定義
IOCSLIB. H	X68000 の IOCS コールの関数の定義
JFCTYPE. H	漢字対応ライブラリ関数の定義
JSTRING. H	漢字対応ライブラリ関数の定義
LIMITS. H	各データ型の有効範囲の定義
MATH. H	算術ライブラリ関数の定義
MOUSE. H	マウス関数の定義
MUSIC. H	FM 音源関数の定義
MUSIC2. H	FM 音源（MIDI 対応）関数の定義
PROCESS. H	プロセス制御関数の定義
SETJMP. H	SETJMP 関数等の定義
SIGNAL. H	SIGNAL 関数等の定義

ファイル名	内　　容
SPRITE. H	スプライト関数の定義
STAT. H	STAT および FSTAT 関数等の定義
STDARG. H	不定型引数の取り出し用マクロの定義
STDDEF. H	標準定数、標準関数用シンボルの定義
STDIO. H	標準入出力関数の定義
STDLIB. H	標準ライブラリ関数の定義
STICK. H	ジョイスティック関数の定義
STRING. H	文字列操作用関数の定義
TIME. H	TIME 関数等の定義
TIMEB. H	FTIME 関数等の定義
UTIME. H	UTIME 関数等の定義
DOSCALL. MAC	Human68k のファンクションコールのヘッダファイル
ERROR. MAC	エラーコードのヘッダファイル
FCNTL. MAC	低水準入出力のヘッダファイル
FDEF. H	BASIC 外部関数用の定数の定義
FEFUNC. H	浮動小数点演算エミュレータ呼び出し用シンボルの定義
IOCSCALL. MAC	X68000 の IOCS コールのヘッダファイル
LIMITS. MAC	各データ型の数値範囲のヘッダファイル
MALLOC. MAC	メモリ管理のヘッダファイル
MATH. MAC	算術演算のヘッダファイル
PROCESS. MAC	プロセス制御のヘッダファイル
STAT. MAC	STAT および FSTAT のヘッダファイル
STDIO. MAC	低水準入出力のヘッダファイル
TIME. MAC	TIME のヘッダファイル

(3) LIB

このディレクトリには、XC コンパイラで使用するすべてのライブラリファイルが格納されています。
ライブラリファイルは、ライブラリアン（LIB. X）で複数のオブジェクトファイルを連結して、1つにまとめたファイルです。

ファイル名	内　　容
CLIB. L	C ライブラリ
BASLIB. L	BASTOC ライブラリ
DOSLIB. L	DOS コールライブラリ
IOCSLIB. L	IOCS コールライブラリ
FLOATFNC. L	浮動小数点ドライバ呼び出しライブラリ
FLOATEML. L	浮動小数点エミュレータライブラリ

(4) BASIC2

このディレクトリには、X-BASIC ver 2.0 の本体、および外部関数が格納されています。

ファイル名	内　　容
BASIC. X	BASIC インタープリタ
BASIC. CNF	BASIC コンフィグファイル
MUSIC2. FNC	FM 音源（MIDI 対応）外部関数
AUDIO. FNC	ADPCM 外部関数
GRAPH. FNC	グラフィック外部関数
MOUSE. FNC	マウス外部関数
STICK. FNC	ジョイスティック外部関数
IMAGE. FNC	イメージ外部関数
SPRITE. FNC	スプライト外部関数

(5) BC

このディレクトリには、BASTOC (X-BASIC から XC へのプログラム変換) の時に使用されるプログラムおよび外部関数の定義ファイルが格納されています。

ファイル名	内　　容
BC. X	BASTOC
AUDIO. DEF	ADPCM 用 DEF ファイル
MOUSE. DEF	マウス用 DEF ファイル
SPRITE. DEF	スプライト用 DEF ファイル
STICK. DEF	ジョイスティック用 DEF ファイル
IMAGE. DEF	イメージ用 DEF ファイル
GRAPH. DEF	グラフィック用 DEF ファイル
BASIC. DEF	BASIC 組み込み関数用 DEF ファイル
MUSIC2. DEF	FM 音源 (MIDI 対応) 用 DEF ファイル
BASIC. CNF	BASTOC コンフィグファイル

(6) SAMPLE

このディレクトリには、いくつかのサンプルプログラムが格納されています。

サンプルプログラムのコンパイル方法やその機能については、SAMPLE. DOC というドキュメントファイルを参照してください。

1.2.3 XC ライブラリディスク

XC ライブラリディスクには、XC システムディスク NO.2 にあるライブラリのソースプログラムが含まれています。

ファイル名	内　　容
CLIB. ARC	C ライブラリのソースファイル
BASLIB. ARC	BASIC ライブラリのソースファイル

1.2 ディスクの内容

CLIB. ARC や BASLIB. ARC のようなアーカイブファイルからファイルをとり出す方法と、内容を確認する方法を説明します。

(1) ソースファイルをとり出す方法

書式

```
A>ar  /a  /x  アーカイブファイル  とり出すファイル
```

例

```
A>ar  /a  /x  clib. arc  cos. s↵
```

CLIB. ARC から COS. S というソースファイルをとり出します。

(2) 内容を確認する方法

書式

```
A>ar  /a  /l  アーカイブファイル
```

例

```
A>ar  /a  /l  clib. arc↵
```

CLIB. ARC に含まれるソースファイルの一覧を表示します。

1.3 ファイル構成

XCコンパイラは、おもにコンパイラ本体、インクルードファイル、ライブラリファイルの3種類のファイルから構成されています。
本節では、これら3つのファイルを中心に説明します。

1.3.1 実行可能ファイル

Human68kの実行可能ファイルは.Xという拡張子をもっています。
以下には、代表的な実行可能ファイルを説明します。

(1) CC. X

CC. Xは、コンパイラを制御する部分とCコンパイラ本体がひとつになったプログラムファイルです。
BASICからCに変換したり、Cのソースファイルをコンパイルして、アセンブラソースファイルやオブジェクトファイルを作成するCコンパイラ本体と、必要に応じてLK. Xを呼び出す、つまりコンパイラ中の各フェーズをつかさどるプログラムからできています。
CC. Xは、LK. Xを呼び出すことにより、実行可能ファイルを作成します。
つまり、コンパイラを実行するには、このCC. Xを呼び出せばよいわけで、Human68kのコマンドモードから"CC"と、それに続くスイッチ指定や、ファイル名指定とともに入力することにより、CC. Xを呼び出します。

(2) AS. X

AS. Xはアセンブラです。
この実行ファイルは、上記のCC. Xでアセンブラソースファイルを出力した場合に、そのあとから、オブジェクトを作成するときに使用します。
C言語プログラムの開発に直接関係しませんが、コンパイラを深く知ることにより、アセンブリ言語と組み合わせて、開発を行うことも可能になります。
なお、アセンブラの詳しい説明は、「アセンブラマニュアル」を参照してください。

(3) LK. X

LK. X は、上記の AS. X でアセンブルされたオブジェクトコードを、他のオブジェクトコードとともに結びつけて、目的とする実行可能ファイルを作成するプログラムです。
一般的にこの LK. X をリンカと呼んでいます。

(4) AR. X

AR. X は、テキストファイル管理プログラムで、アーカイバと呼びます。
AR. X では、テキストファイルの作成、変更、削除といったテキストファイルに関する管理を行います。

(5) LIB. X

LIB. X は、オブジェクト管理プログラムで、ライブラリアンと呼びます。
LIB. X では、オブジェクトファイルの作成、変更、削除といったオブジェクトファイルに関する管理を行います。

(6) BC. X

BC. X は、BASIC から C 言語に変換するトランスレータです。
詳しい説明は、「第 5 章　XBAStoC BASIC プログラムコンバータ」で行います。

1.3.2 インクルードファイル

インクルードファイルは、C 言語の# include という命令を使って、プリプロセッサが、とり込むことができるテキストファイルです。
このインクルードファイルは、. H という拡張子をもったファイルで構成されており、ユーザープログラム中の先頭に # include ＜インクルードファイル名＞ と宣言しておけば、コンパイラのプリプロセッサにより、インクルードファイル名にあたる部分をユーザープログラムに展開してくれます。
標準で提供されるインクルードファイルは、「1.2　ディスクの内容」で示したようなファイルにより構成されていますが、XC コンパイラの利用者も、独自のインクルードファイルを作成して登録できます。
このような機能は、C 言語の特長の 1 つでもあり、プログラムの簡素化と効率のよいプログラム展開をするうえで重要な機能です。

1.3.3 ライブラリファイル

ライブラリファイルは.Lという拡張子をもったファイルで構成されています。

ライブラリもインクルードファイルと同様に、C言語の大きな特長の1つになっています。

もともと、C言語では入出力関係などをサポートしておらず、マシンに依存する部分は、すべてライブラリというオブジェクトファイルの集まりによって提供されます。

「1.3.1　実行可能ファイル」で紹介したリンカ（LK.X）によって、ユーザープログラムが必要とするライブラリ中の関数が、このライブラリファイルから抜きとられて、リンクされます。

XCコンパイラによるプログラム開発を行ううえで、利用者自身のライブラリファイルを作成して管理することにより、開発の効率が上がります。

(1) CLIB. L

C言語専用のライブラリです。

このライブラリのソースファイルは、XCライブラリディスクにCLIB.ARCという名前で入っています。

(2) BASLIB. L

X-BASIC専用のライブラリです。

このライブラリのソースファイルも、XCライブラリディスクにBASLIB.ARCという名前で入っています。

(3) DOSLIB. L

Human68kのファンクションコールを呼び出すためのライブラリです。

(4) IOCSLIB. L

X68000のIOCSコールを呼び出すためのライブラリです。

(5) FLOATFNC. L

浮動小数点演算ドライバを呼び出すライブラリです。

これを使用すると、実行ファイルがコンパクトになり、コプロセッサボードを利用することもできます。

作成した実行ファイルを動作させるためには、FLOAT2. X または FLOAT3. X が必要です。

(6) FLOATEML. L

浮動小数点演算を行うライブラリです。

これを使用すると、FLOAT2. X を必要としない実行ファイルが作成できます。

1.4 インストールと開発環境の設定

本章では、XCコンパイラのインストール方法を説明します。
インストールを行わないと、XCコンパイラを使用することはできません。
XCコンパイラのインストールには、INSTALL. BATを利用するオート・インストールと、ユーザーがコマンドシェル上からCOPYコマンドやMKDIRコマンドなどを利用して、オリジナルのシステムディスクよりファイルをコピーするマニュアル・インストールの2種類があります。
ほとんどの場合、オート・インストールでインストールできるので、最初にオート・インストールを試してみてください。
以下では、オート・インストールとマニュアル・インストールを分けて説明します。

1.4.1 オート・インストール

本項では、INSTALL. BATを用いたオート・インストールについて説明します。
また、この場合インストールするデバイスによりその方法が違うので、フロッピーディスクとハードディスクに分けて解説します。

フロッピーディスクの場合

XCコンパイラver 2.0は、その機能が大幅に拡張されたのでシステムディスク1枚では収まりきらず、X-BASICやXCを使うために、ディスク2枚分のユーティリティが必要となります。
そのため、オリジナルのディスク構成では、システムディスクだけでフロッピーディスクドライブ2台を占有してしまい、ユーザーの作業用ディスクが使用できません。
そこで、作業用ディスクを使用できるよう、システムの再構成を行うのがインストールプログラムの役割です。
このインストールプログラムにより、XCコンパイラは次の3枚のディスクにまとめられます。

1.4 インストールと開発環境の設定

(1) Human68k 起動ディスク

Human68k を起動し、デバイスドライバを組み込み、コマンドシェルを立ち上げるためのディスクです。

このディスクは、システム起動後にランタイムディスクと交換します。

このディスクにはエディタやソースコードデバッガなどのツールが納められています。

これらをお使いになる場合はランタイムディスクと交換します。

(2) ランタイムディスク

XC コンパイラの本体や、X-BASIC やリンカなどのユーティリティが格納されたディスクです。

システム起動後に Human68k 起動ディスクと交換し、以後システムディスクとして使用します。

通常、このディスクはAドライブとなります。

(3) 作業用ディスク

ユーザーが個人で作成したプログラムのソースファイルやオブジェクトファイルなどを格納するディスクです。

通常、このディスクはBドライブになります。

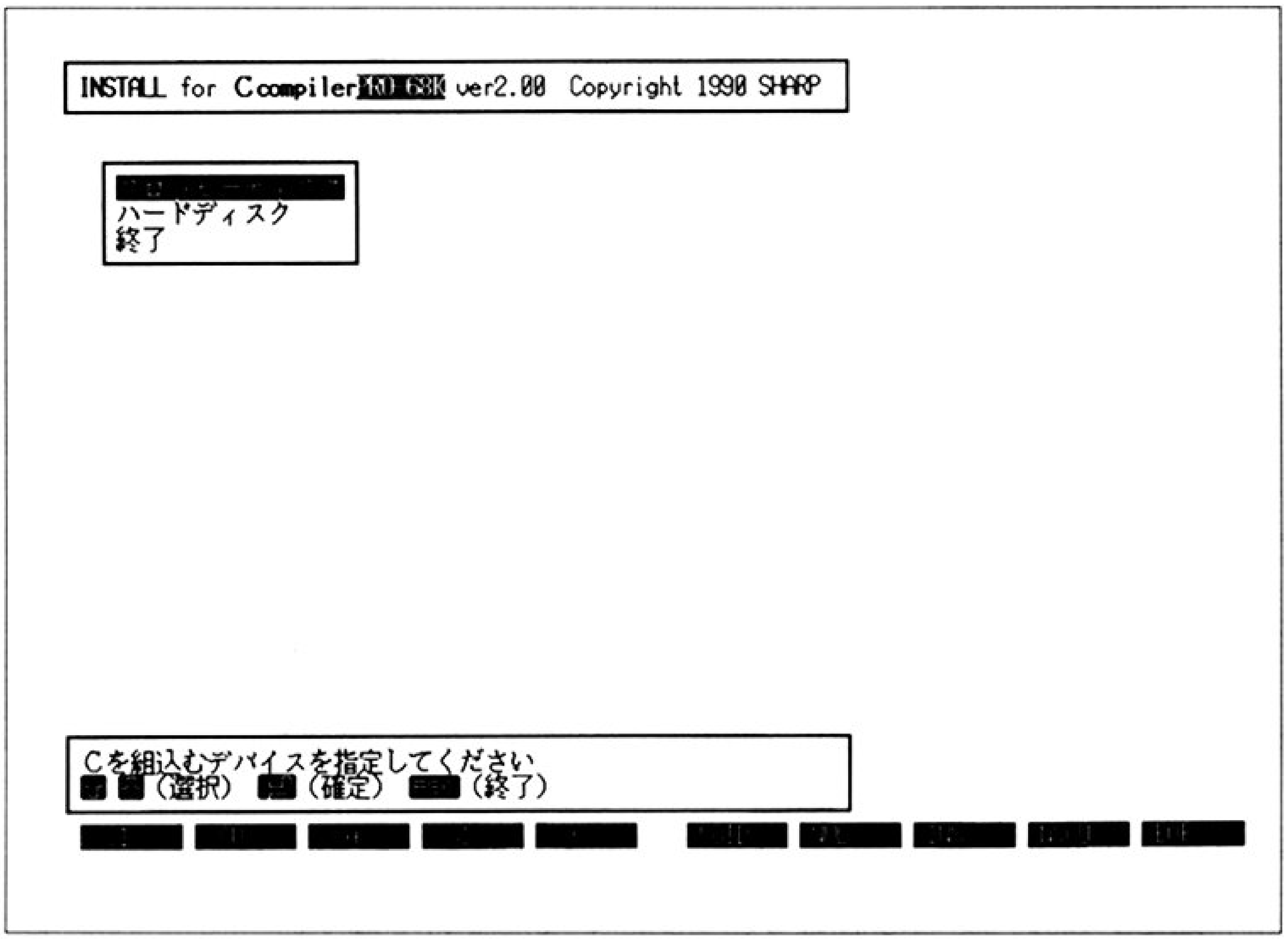
つぎに、インストールプログラム INSTALL. BAT を使用したシステムのインストール方法について説明します。

①まず、インストールの際に使用するディスクを2枚作成します。
　このディスク2枚は、FORMAT コマンドでフォーマットした空きディスクを用意します。
　フォーマットの方法は、「1.1　バックアップディスクの作成」を参照してください。

② X68000 の電源を ON にして、バックアップした XC システムディスク No.1 をドライブ0にセットしてください。

③ Human68k の起動メッセージの後に、「C コンパイラの組み込みを始めますか [Y／N]」というメッセージが表示されたら、'Y'をキーボードより入力します。
　インストールするつもりがないときは、ここで'N'を入力すると、コマンドシェルに戻ります。

④インストールプログラムが読み込まれ、次のようなメニュー画面が表示されます。



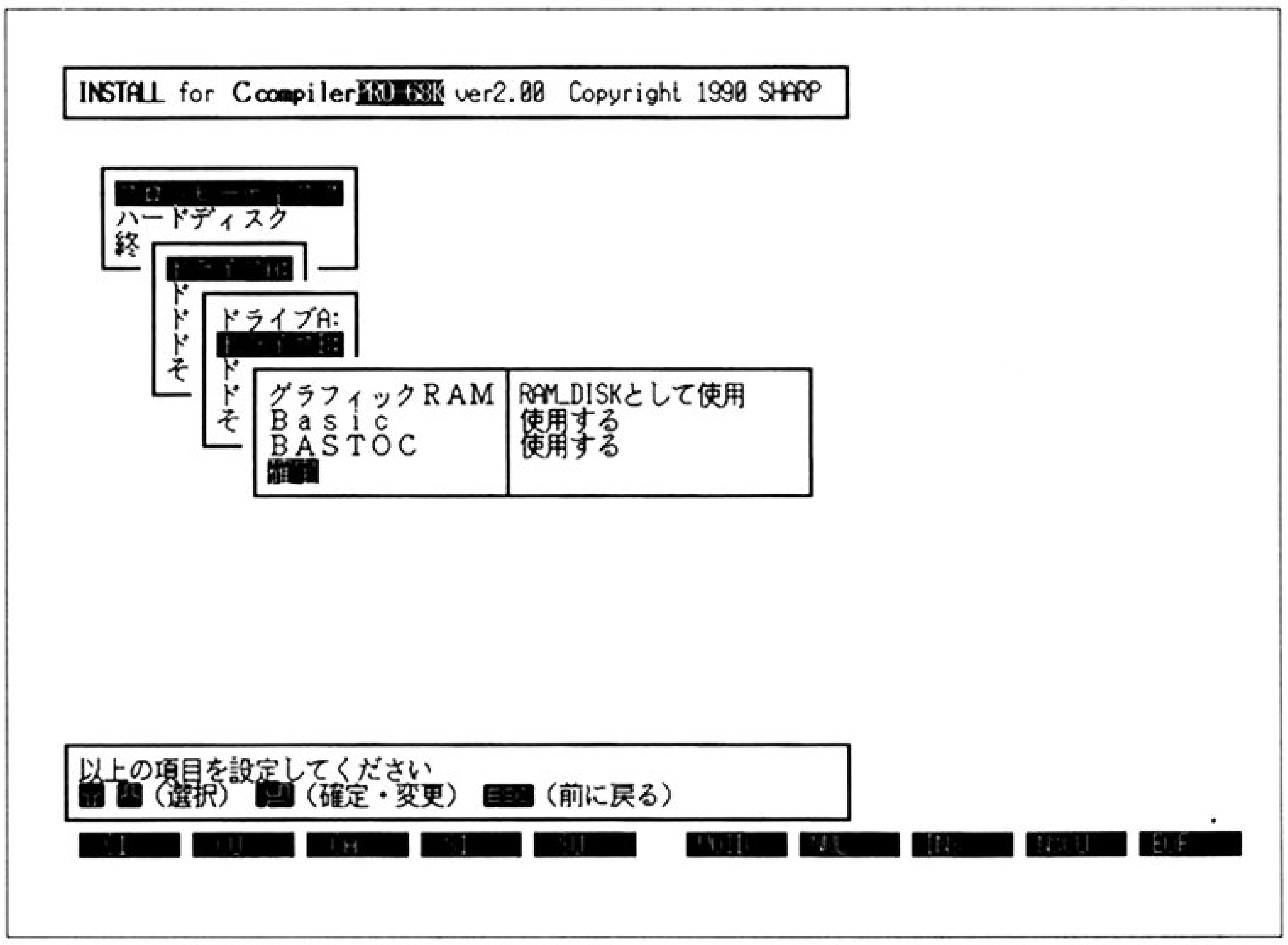
ここでは、XC コンパイラを組み込むデバイスを選択します。
[↑]、[↓]キーで黄色のカーソルを「フロッピーディスク」に合わせて、リターンキーを押してください。

⑤フォーマット済みのフロッピーディスク２枚の用意ができているかどうかを聞いてくるので、'Y'を入力してください。
もし、①で用意していなければ、'N'を入力すればコマンドシェルに戻ります。

⑥次に、コピー元のドライブ名を聞いてくるので、現在システムディスクがセットされているドライブ（通常はＡドライブ）を選択してください。

⑦次に、コピー先のドライブ名を聞いてくるので、現在空いているドライブ（Ｂドライブ）を選択してください。

⑧ここで、次のように RAM ディスクや BASIC などを使用するかどうかを聞いてきます。



必要に応じて各項目を↑、↓キーとリターンキーで設定し、終わったら「確定」にカーソルを合わせ、リターンキーを押します。
なお、「使用しない」と設定した項目は、ランタイムディスクから省かれるため、その分ランタイムディスクの空き容量が増加し、インストール終了後に別のコマンドやユーティリティをコピーすることができます。

⑨ Human68k 起動ディスクを作成するため、XC システムディスク No.1 からのコピーを行うので、フォーマット済みの空きディスクを指定されたドライブにセットし、'Y'を入力してください。
するとコピーが開始されます。
※コピー中は、キーを押さないでください。

⑩ Human68k 起動ディスクを作成し終わると、次はランタイムディスクの作成です。
XC システムディスク No.2 とフォーマット済みの空きディスクを、指定されたドライブにセットしてください。
その後、'Y'を入力するとコピーが始まります。

⑪XC システムディスク No.2 からランタイムディスクへのコピーが終了すると、次は、XC システムディスク No.1 からランタイムディスクへのコ

ピーを行います。
XC システムディスク No.1 を指定されたドライブにセットしてから、'Y' キーを入力してください。
コピーが開始されます。

⑫コピーが終了すると、リターンキーの入力待ちになります。
ここでリターンキーを入力すると、フロッピーディスクへのインストールは終了になります。

⑬最後に、今作成した起動ディスクをドライブ0に入れることを促すメッセージが画面に表示されます。
ここで、'Y'を入力するとリセットされて Human68k が起動されます。
'N'を入力するとコマンドシェルに戻ります。

このようにして作成したディスクでシステムを起動するには、次の手順が必要です。

① X68000 の電源を ON にし、ドライブ0に Human68k 起動ディスクをセットします。

② Human68k が起動すると、ランタイムディスクと差し替えることを促すメッセージが画面に表示されます。
そこで、ドライブ0にセットされている Human68k 起動ディスクとランタイムディスクを差し替え、ドライブ1に作業用ディスクをセットします。

これで、XC コンパイラがフロッピーディスクベースで使用可能になります。

ハードディスクの場合

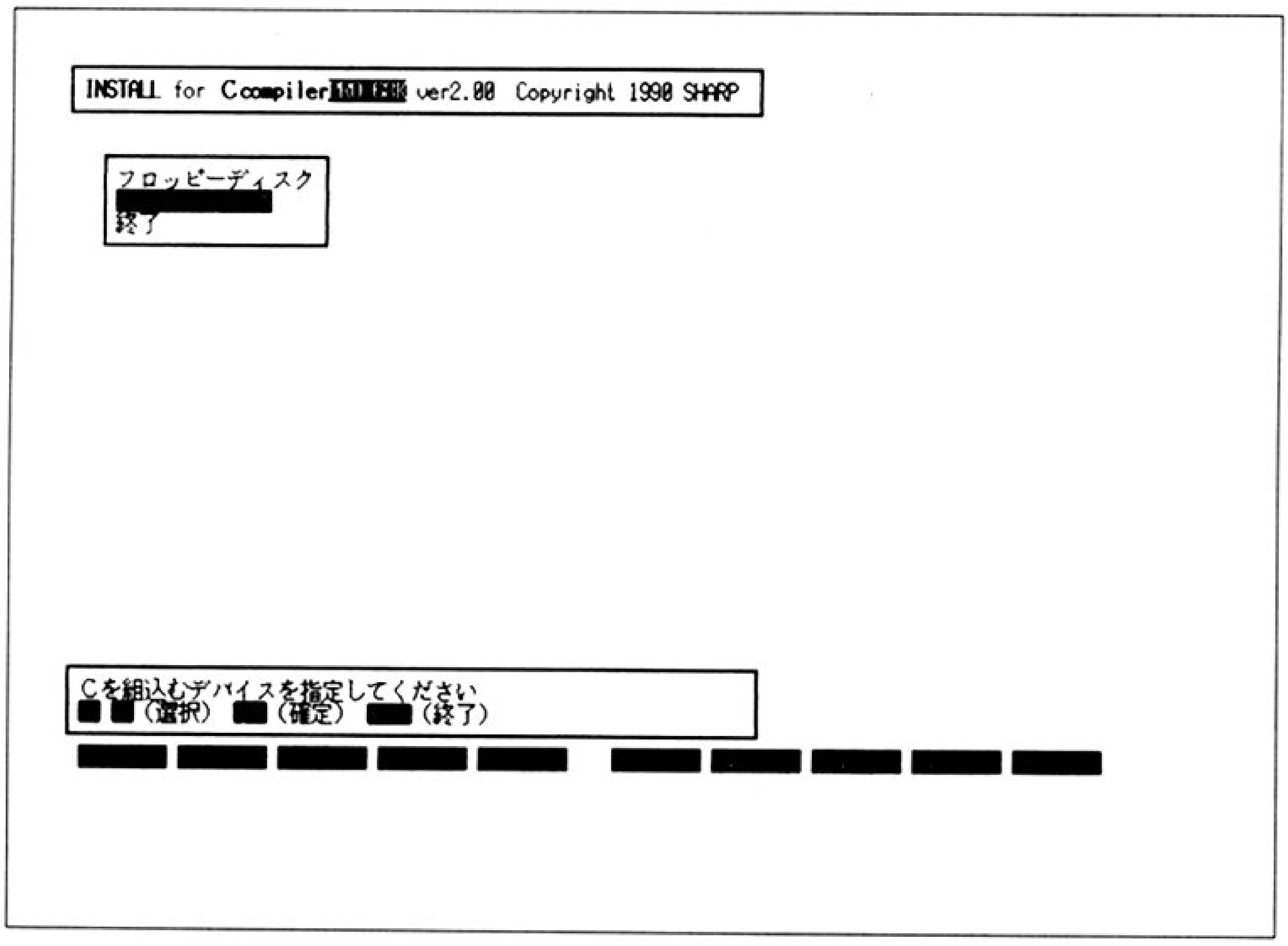
XC コンパイラ ver 2.0 を、インストールプログラムを用いてハードディスクにインストールするのは、フロッピーディスクにインストールするのに比べるとそれほど難しくありません。
ただ、ハードディスクには最低でも 2M バイトの空き容量が必要であることに注意してください。
インストールプログラム INSTALL. BAT を使用したシステムのインストール方法について説明します。

1.4 インストールと開発環境の設定

①ハードディスクが内蔵でなく増設されている場合は、まずハードディスクの電源をONにします。
つぎにバックアップしたXCシステムディスクNo.1をドライブ0にセットし、[OPT.1]キーを押しながら、X68000の電源をONにしてください。
これにより、ドライブ0にセットされているフロッピーディスクより強制的にHuman68kが起動します。
なお、[OPT.1]キーはHuman68kの起動メッセージが表示されるまで押し続けてください。

② Human68kの起動メッセージの後に、「Cコンパイラの組み込みを始めますか［Y／N］」というメッセージが表示されたら、'Y'をキーボードより入力します。
インストールするつもりがないときは、ここで'N'を入力すると、コマンドシェルに戻ります。

③インストールプログラムが読み込まれます。



ここでは、XCコンパイラを組み込むデバイスを選択します。
[↑]、[↓]キーで黄色のカーソルを、上の画面のように「ハードディスク」に合わせて、リターンキーを押してください。

④コピー先のハードディスクのドライブ名を聞いてくるので、XCコンパイラをインストールするハードディスクのドライブ名を選択してください。

⑤選択されたハードディスクの空き容量が2Mバイト以上あるかどうかチェックします。
もし、空き容量が足りなければ、インストールできないので、コマンドシェルに戻ります。

⑥XCコンパイラ本体（CC. X）やライブラリ、ヘッダファイルなどを格納するディレクトリを設定します。
注意すべきことは、ここで指定できるディレクトリは、ルートディレクトリの１つ下のサブディレクトリだけ、ということです。
つまり

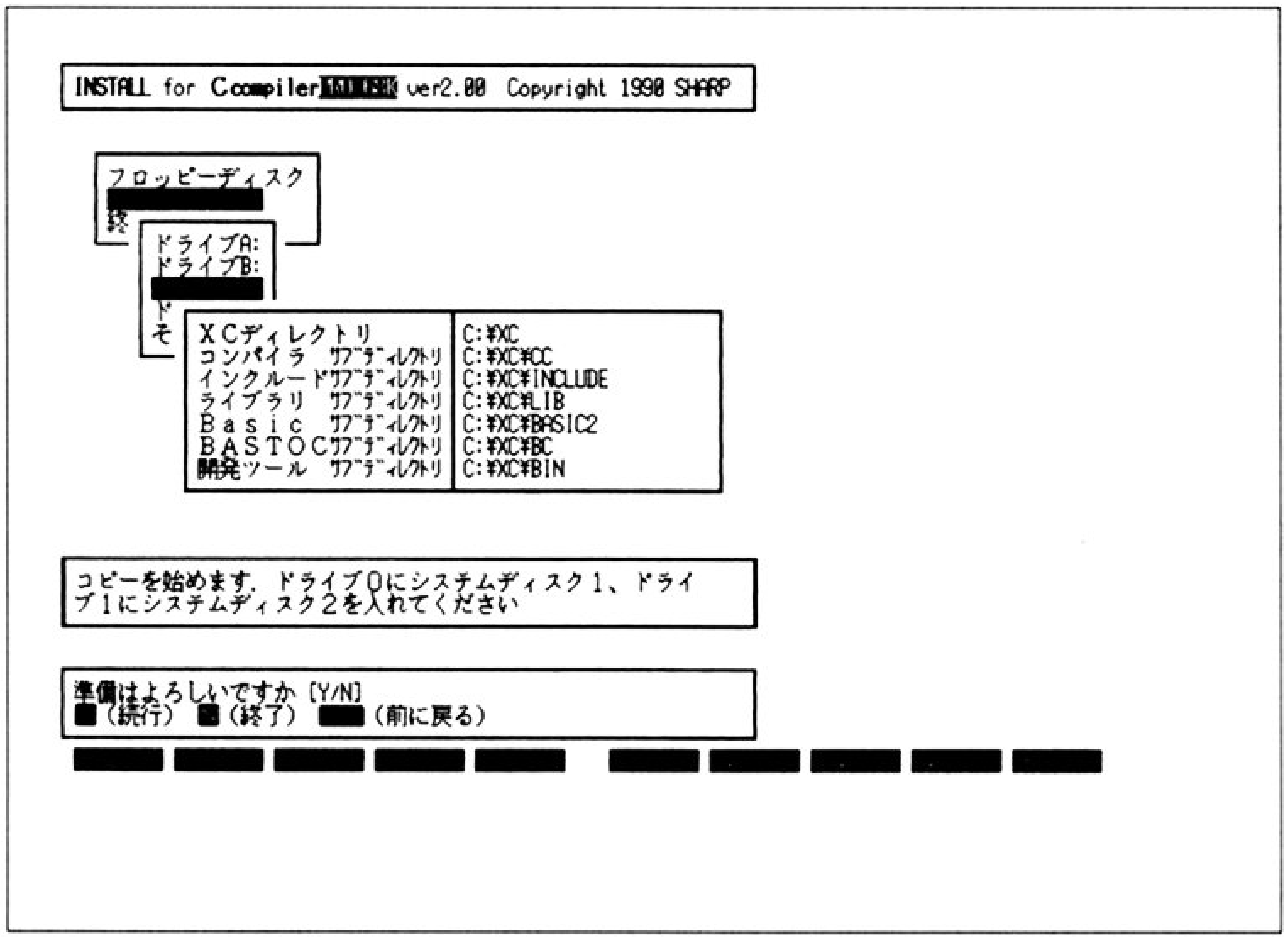
```
C：¥XC_ver2
```

は許されますが、

```
C：¥XC¥ver200
C：¥BIN¥XC
```

といった指定はできません。
もし、このようなサブディレクトリに格納したい場合は、マニュアルでインストールする必要があります。
なお、このときすでに存在するディレクトリを指定した場合は、上書きするかどうか聞いてくるので、上書きしていいなら'Y'を、そうでなければ'N'を入力してください。

⑦ディレクトリの指定が終了すると、次のように作成されるサブディレクトリが表示され、コピーの準備ができているかどうか聞いてきます。



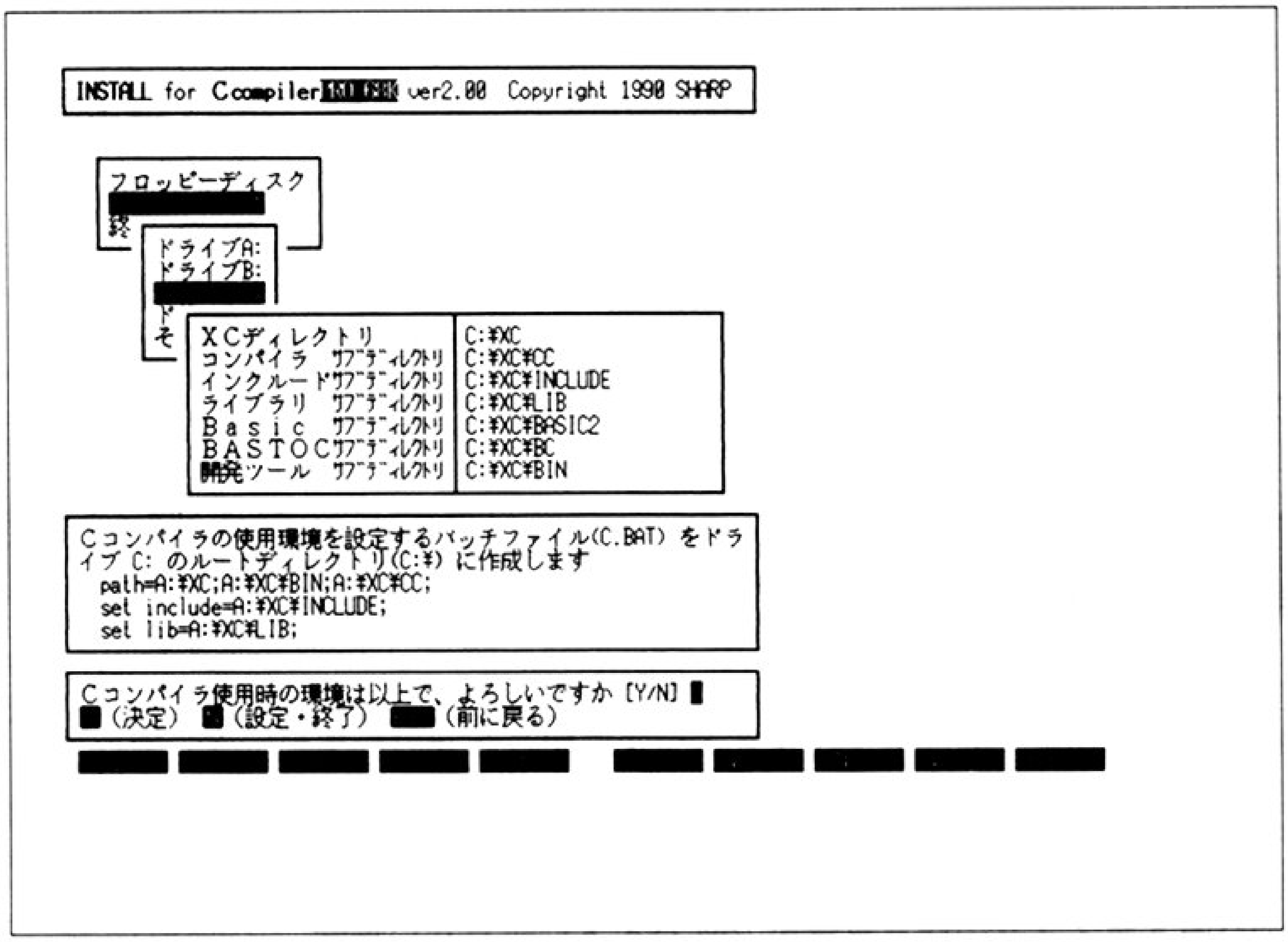
画面の指示通りにXCシステムディスクNo.1、No.2を各ドライブにセットしてください。

システムディスクをセットし終わったら、'Y'を押してください。

コピーが開始されます。

もし、ここで'N'を押すと、インストールは中断されコマンドシェルに戻ります。

⑧コピーが終了すると、Cの環境設定を行うバッチファイルC. BATの内容を表示します。



C. BAT の内容を変更する必要のないときは、'Y'を入力してください。
C. BAT の内容を変更したい場合は、'N'を押してください。
なお、ここで変更できるのは、C. BAT で設定されるコマンド検索パスやライブラリパス、インクルードパスのドライブ名だけです。
C. BAT は、指定したハードディスクのルートディレクトリにコピーされます。
そのため、必ずコマンド検索パスにそのハードディスクのルートディレクトリを指定しておいてください。
指定しな場合、C. BAT を実行できないことがあります。

⑨C. BAT のコピーが終了すると、リターンキーの入力待ちになります。
ここでリターンキーを入力すると、ハードディスクへのインストールは終了になります。
ここで'Y'を入力するとリセットされ、'N'を入力するとコマンドシェルに戻ります。

こうしてインストールした XC コンパイラを使用するには、ハードディスクより Human68k を起動し、コマンドシェルが立ち上がったところで、

```
A>C ⏎
```

としてください。
すると、C. BATが実行され、コマンド検索パスやインクルードパス、ライブラリパスが設定されます。

1.4.2 マニュアル・インストール

ほとんどの場合、オート・インストールで間に合うのですが、次の場合オート・インストールを利用することはできません。

(1)ハードディスクの空き容量が2Mバイトに足りない場合
(2)ハードディスクにインストールする場合、XCコンパイラ本体（CC. X）を格納するサブディレクトリを、ルートディレクトリの２つ以上深いサブディレクトリに作成したい場合

この場合は、ユーザーがCOPYコマンドやMKDIRコマンドなどをコマンドシェル上で実行して、必要なファイルをハードディスクにコピーしなければなりません。
これをマニュアル・インストールといいます。
マニュアル・インストールのときに注意すべき点を以下に示します。

① Human68kやコマンドシェルなどのシステムファイルのバージョンを2.0以上にする。

② Human68k起動時に浮動小数点エミュレータFLOAT2. Xを組み込むよう、CONFIG. SYSを設定する。

③コマンド検索パスやCCのパス、ライブラリパス、インクルードパス、BCのパスをAUTOEXEC. BATの中で設定する。
コマンド検索パスはPATHコマンドで、CCのパスは環境変数cc、ライブラリパスは環境変数lib、インクルードパスは環境変数include、BCのパスは環境変数bcにより設定できます。

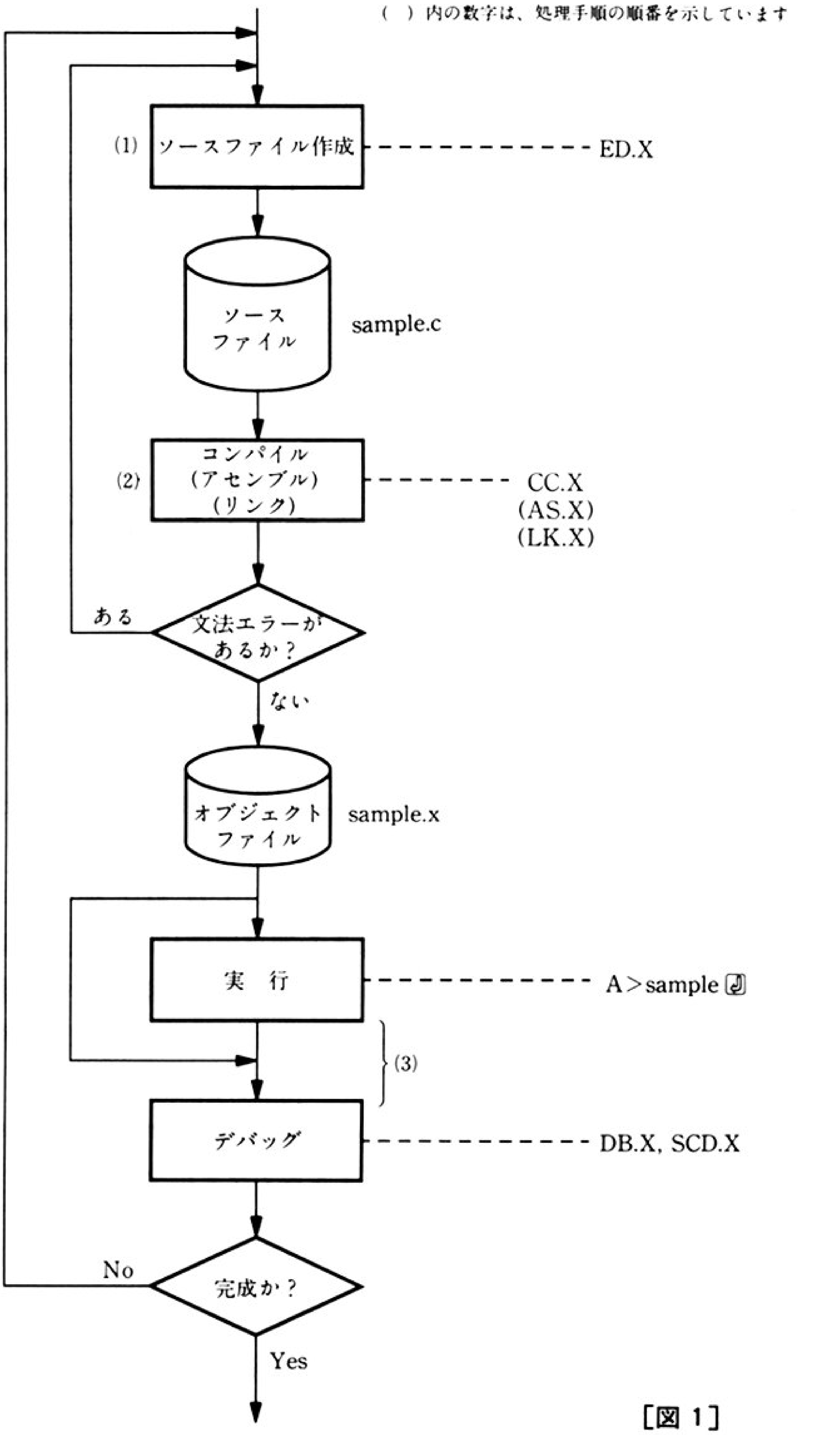
1.5 プログラム開発の流れ

XCコンパイラは、ソースコードからオブジェクトコードを生成する一種のトランスレータですが、プログラム開発を行ううえで、コンパイラをとりまく環境も理解しなければなりません。
前節では、コンパイラが動作するための環境を中心に説明しましたが、本節では、プログラム開発の一連の流れを中心に、プログラム開発に必要なユーティリティなどについて説明します。

1.5.1 ソースプログラムの作成から実行まで

ソフトウェアの開発では、使用する言語によってさまざまな開発方法がありますが、大きく分けると、(1)ソースプログラムを作成し、(2)コンパイルして、(3)デバッグする、という3つのフェーズに分けることができます。
もっとも、BASICのようなインタープリタ言語では、このコンパイルという作業がなくなるわけです。
プログラム開発では、このように使用する言語（ここではXC）を中心にソースプログラムを作成する時点から、どのような開発環境があり、どのようにして実行可能なプログラムを作成するのかを知っておく必要があります。
Human68kオペレーションシステムで、実行可能なプログラムを作成するまでの過程には一定の手順があります。
これは大きく分けて、(1)ソースプログラムの作成(ED. X)　(2)コンパイル(CC. X)　(3)デバッグ(DB. X、SCD. X)という3ステップになっています。
各ステップには、その手順に適したツールが用意されています。
（　）内は対応するツールの名称です。
XCコンパイラを使用してのコンパイルの過程では、ソースコードをコンパイルして実行可能ファイルまでいっきにコンパイルしたり、またアセンブラソースコードまでを出力するというように、XCコンパイラのスイッチ指定により、さまざまな開発方法が可能です。
通常の開発では、実行可能ファイルまでを生成するようにします。
次に、プログラム開発がどのようなものかを手順に沿って説明します。

1.5 プログラム開発の流れ

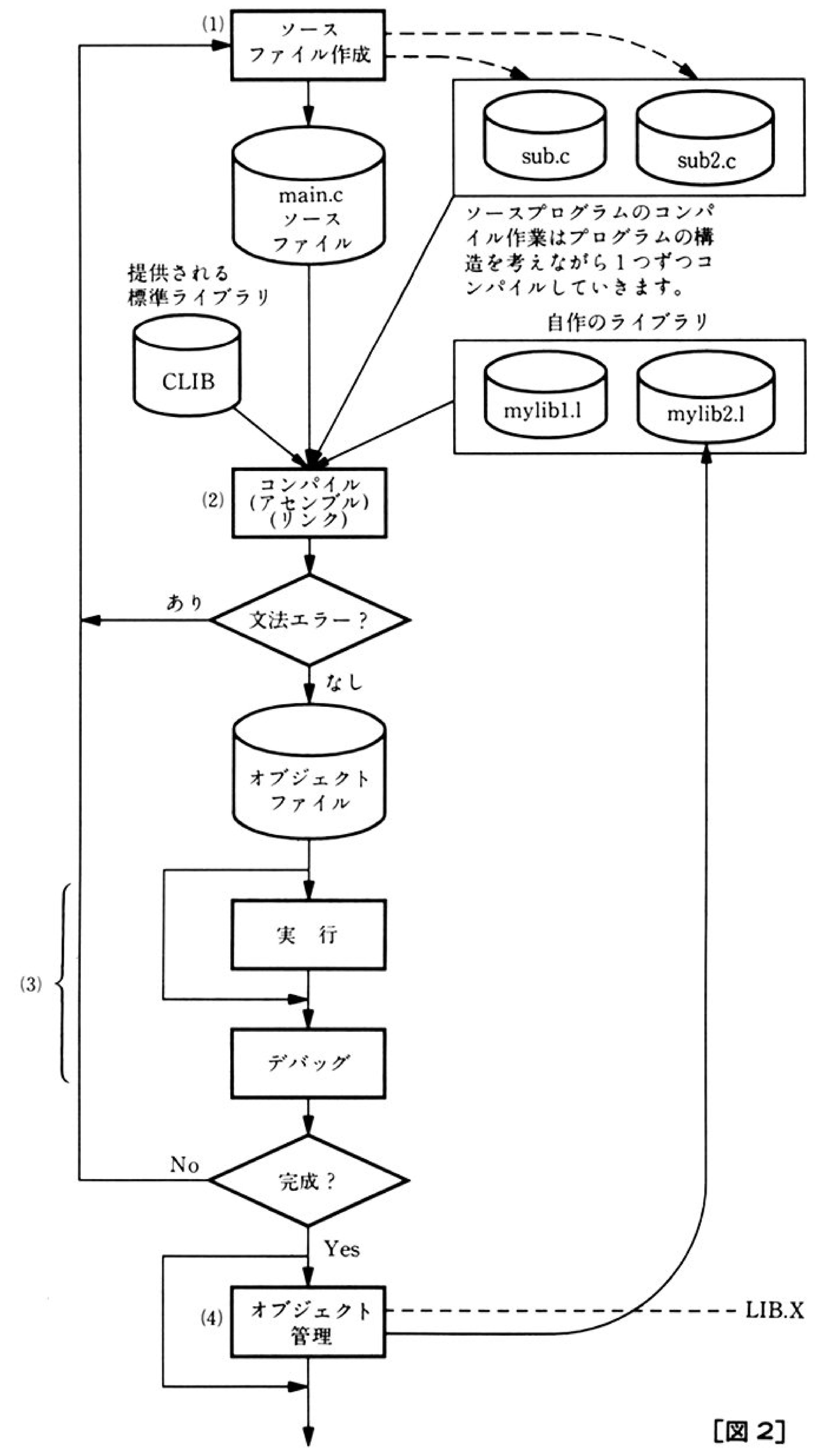


[図 1]

通常のひとつのプログラムについては、上記で示したような過程によりプログラムを完成しますが、C言語によるプログラム開発において、目的とするソフトはメインとなるプログラムと複数のサブプログラムを管理しながら開発を行わなければなりません。

1.5　プログラム開発の流れ

次に示すフローチャートは、複数のプログラムの管理を前提に、開発の流れを示したものです。



［図２］

1.5 プログラム開発の流れ

図2は、図1に比べてオブジェクトの管理という作業が増えています。
それでは、各図に示されたフェーズの1つ1つを説明しましょう。

(1) ソースプログラムの作成

ソースプログラムは.Cという拡張子をもったファイル名で、ディスク上に作成します。
X68000では、EDというエディタを使い、このソースプログラムの作成作業を行います。
EDは、フルスクリーンエディタといって、カーソルをスクリーン上で自由に動かせることができ、効率のよい編集が可能です。
EDは、次のように起動します。
詳しい説明は、Human68kユーザーズマニュアルを参照してください。

```
ED [<ドライブ名：>][<ファイル名>]
```

(2) コンパイル

ソースファイルは、そのままではただのテキストファイルです。
このC言語で記述したプログラムを、X68000が実行できるような実行可能ファイルに変換しなければなりません。
これをコンパイルといい、C言語の他に代表的な言語としてはCOBOLやFORTRANなどがあります。
また皆さんに馴染みの深いBASICは、インタプリタ言語といい、ソースプログラムを1命令ずつ機械語に翻訳しながら実行します。
この他、インタプリタ言語にはLISP、PROLOGなどがあります。
コンパイラとインタプリタには、それぞれメリット、デメリットがあり、いちがいにどちらがよいというわけにはいきませんが、使用目的によって使いかたが異なってきます。

コンパイラ……実行速度が早く、アプリケーションやシステムの開発に向いているが、初心者には向いていない。
インタプリタ…コンパイラに比べて実行速度がやや落ちるが、開発が比較的容易にでき、初心者には向いている。

このように、コンパイラ、インタプリタにはそれぞれの特長がありますが、最近ではインタプリタもかなり速度が向上してきており、きめの細かい処理

ができるようになってきました。

なお、X68000ではBASICからC言語に変換するBASTOCが用意されており、BASICとC言語のギャップを解消しています。

XCコンパイラは次のように起動します

CC［<スイッチ>］<ファイル名>［<ファイル名>］

このように１つのコマンドラインに、コンパイラの各フェーズで必要になる制御を、すべて指定します。

また、ここで単に'CC'と入力すると、ヘルプメッセージが表示されます。

ヘルプメッセージには、スイッチの説明が表示されます。

```
X68k XC Compiler v2.00 Copyright 1987,88,89,90 SHARP/Hudson
使用法:CC [スイッチ] ファイルリスト
/A   警告レベルの指定
/B   C言語ソースファイル(.c)のみの作成
/C   コメント行の保存
/D   マクロ定義
/E   識別子の最大長の指定
/Fc  リンクの抑制
/Fd  アセンブラソースファイル(.s)の出力ファイル名の指定
/Fo  オブジェクトファイル(.o)の出力ファイル名の指定
/Fs  アセンブラソースファイル(.s)のみの作成
/Fx  実行可能ファイル(.x)の出力ファイル名の指定
/G0  シンボルの最大個数の指定(パーザ部)
/Ga  シンボルの最大個数の指定(AS部)
/Gb  ハッシュバッファの最大個数の指定(BC部)
/Gc  スタック検査の指定
/Gh  ヒープサイズの設定
/Gp  シンボルの最大個数の指定(プリプロセッサ部)
/Gs  スタックサイズの設定
/I   インクルードパスの指定
/J   char型をunsigned char型とする
/Na  32kバイト以上のauto変数の使用許可
/Nf  浮動小数点演算ライブラリの変更
/Nl  ライブラリパスの指定
/Ns  ソースコードデバッガ(SCD.X)用コードの生成
/O   プログラムの最適化
/P   プリプロセッサの出力をファイルへ出力
/Q   同一コード文字列の圧縮
/T   作業パスの指定
/U   マクロ定義の取り消し
/V   プリプロセッサの出力を標準出力へ出力
/W   BASICライブラリの使用フラグ
/X   定義済み識別子__STDC__を未定義にする
/Y   DOS/IOCSライブラリの使用フラグ
```

(3) 実行とデバッグ

コンパイルを終了すると、実行可能なプログラムが作成されます。
実行可能なプログラムは、たとえばtest.cというファイルをコンパイルすると、test.xというファイル名で出力されます。
ここで、いきなり'A>'プロンプトから、'test ⏎'と入力して実行させても、いっこうにさしつかえありません。
簡単なプログラムの場合は、1回のコンパイルで完成してしまう場合もありますが、通常のプログラム開発では、数回デバッグ作業を繰り返した後にプログラムを完成します。
実行可能なプログラムのデバッグ作業は、DB. XまたはSCD. Xというデバッガを使って作業します。

実行時のスピードについて

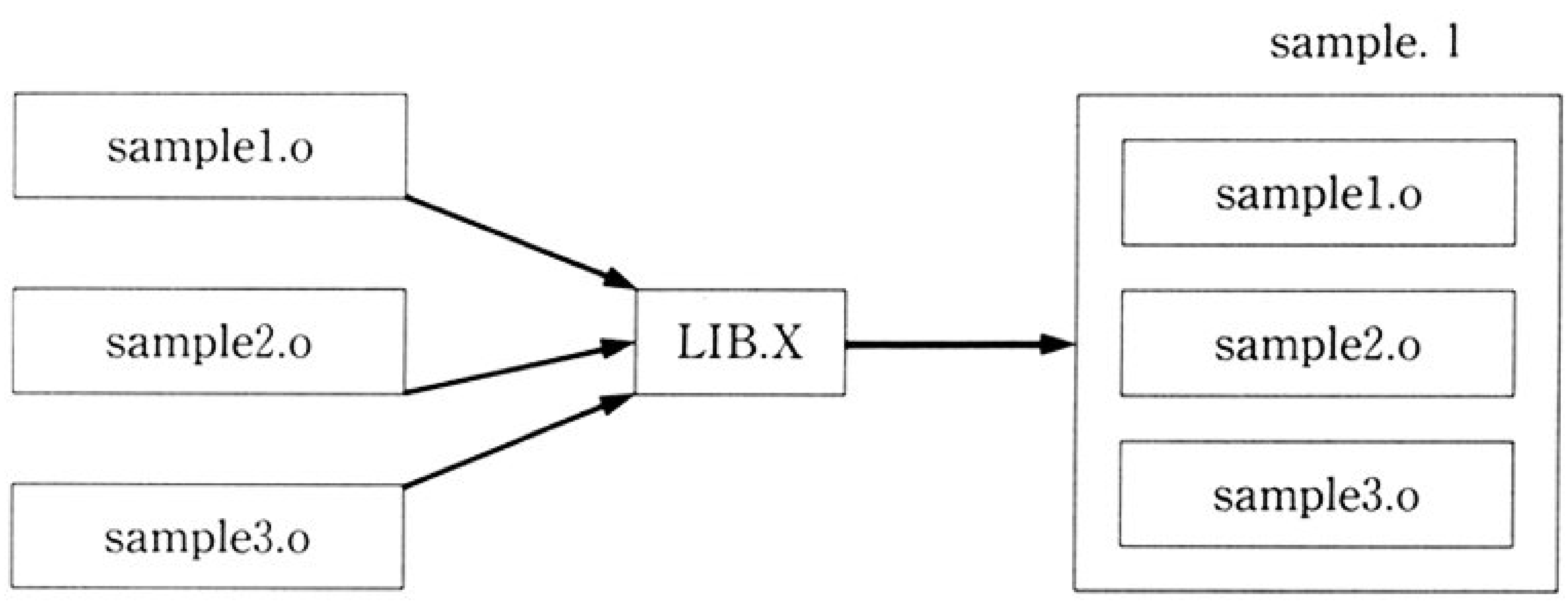
XCシステムディスクでは、起動時にFM音源用のデバイスドライバ(OPMDRV2. X)をOSに組み込んでいます。
その結果、コンパイラで作成されたプログラムを実行中に常時FM音源の割り込みがかかり、実行スピードが遅くなります。
このため、FM音源を使用しない場合に限り、CONFIG. SYSからOPMDRV2. Xを削除し、実行プログラムのスピードを向上させることができます。

(4) オブジェクトの登録と管理

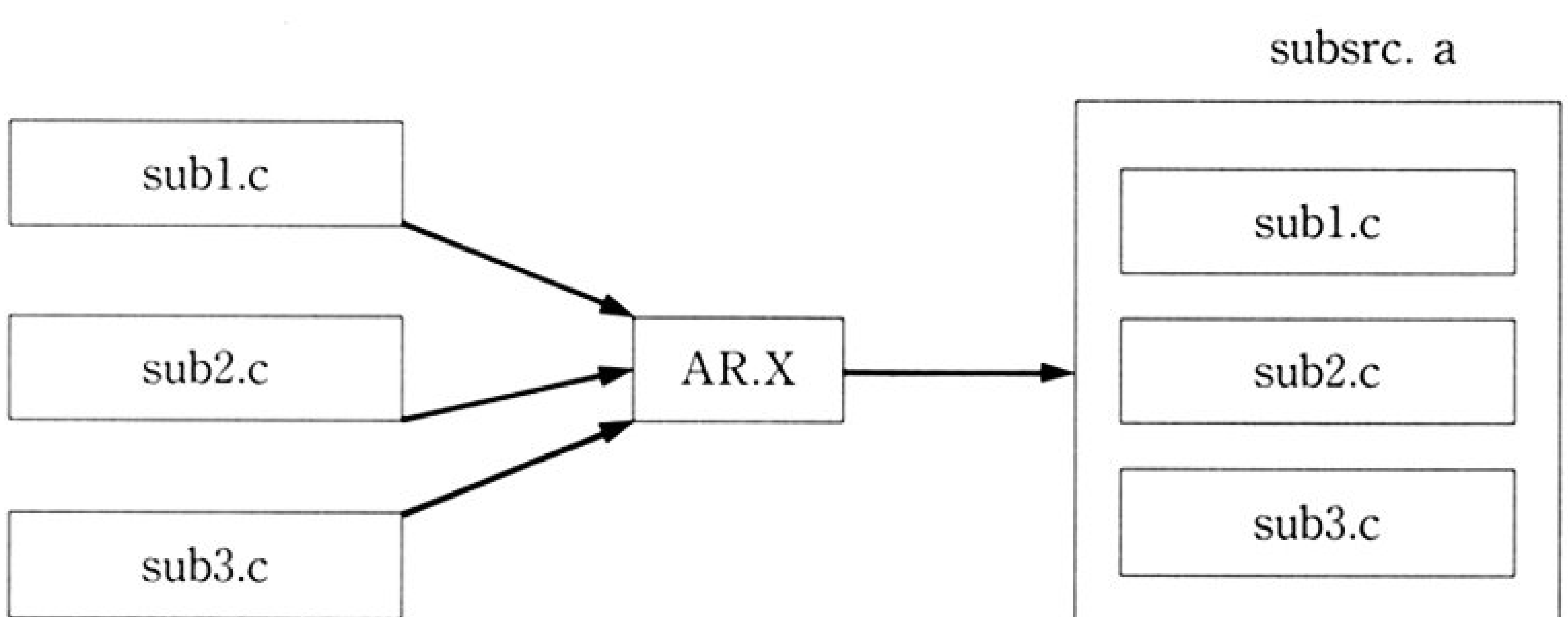
C言語を使って開発するうえでもっとも特長的なものは、このオブジェクトの管理です。
通常、BASICなどではサブルーチンは行番号で管理され、1つのファイルに納められています。

C言語での開発では、分割コンパイルといって、各機能ごとにプログラムを分割して開発を進めていきます。
完成済みの各プログラムは、XCコンパイラ(実際にはリンカ)によって結びつけられ、1つのプログラムにまとめ上げられます。
この開発で、オブジェクトの管理が必要になります。
プログラムが数個程度に分割されている、比較的小規模なプログラム開発の場合はさほど問題になりませんが、分割されたプログラムの数が多くなると、ファイルの数も多くなり、開発の効率も落ちてきます。
そこで、プログラムのオブジェクトを機能分野ごとにまとめて、1つのファイルとして管理するようにします。

たとえば、グラフィックス関係の関数は、すべて１つのファイルにしまい込みます。
そして、リンクするときにメインプログラムが必要としているプログラムだけを、そこから抜きとってリンクするというように、オブジェクトの集合ファイルを作成するわけです。
このようなオブジェクトファイルの集合ファイルを、ライブラリファイルと呼びます。
ライブラリファイルを管理するのがLIB. Xで、ライブラリアンと呼びます。ライブラリアンは、このような分割化の長所と一本化の長所をともに活かすためのツールです。
分割コンパイルしたオブジェクトファイルを拡張子. Lで示されるライブラリファイルに登録します。
ライブラリファイルには、複数のファイルを登録できますから、ディレクトリ上からは、あたかも１つのファイルとして管理することができます。
ライブラリファイルを作成して、開発の効率化をはかってください。
なお、ソースファイルの管理にはAR. Xを使用してください。



［オブジェクトファイルのライブラリファイル］



［ソースファイルのアーカイブファイル］

第2章
コンパイル

起動方法
スイッチによる制御
アセンブル
リンク

CC は、C 言語で記述したソースファイルをコンパイルするコマンドです。第 1 章でもコンパイラについての概要は説明しましたが、本章では、このコンパイラをこれから使用されるかたのために、コンパイラである CC. X の説明と起動方法、さらに、コマンドラインの指定方法を説明します。

2.1 起動方法

コンパイラとはCC.Xのことをいいます。
通常のコンパイル作業では、このCC.Xを使ってコンパイルを行います。

2.1.1 コマンドラインの指定

コマンドラインとは'CC'から始まるコマンド行のことをいい、コンパイルで必要なすべての指示をこの行で指定します。
まずは、以下のコマンド行を見てください。

```
B>cc sample.c ⏎
```

と、入力することにより、sample.cというソースファイルはコンパイルされ、アセンブラ、そしてリンカをへて最終的にsample.xという実行可能ファイルを出力するわけです。
これがコマンドラインの基本的な使いかたですが、コマンドラインではこの他にもスイッチ指定などにより、コンパイラを多角的に使用することができます。
たとえば、次のコマンドラインを見てください。

```
B>cc ／Fc sample.c ⏎
```

このコマンドラインは、／Fcというスイッチ指定によりリンクを行わないで、sample.oというオブジェクトファイルを出力しなさいという意味です。
スイッチなどについては、このあと詳しく説明することにします。

2.1　起動方法

(1)コマンドラインの書式

```
CC [<スイッチ>] <ファイル名> [<ファイル名>……]
```

(2)ファイル名の指定

ファイル名は、OSであるHnman68kに依存しますが、全角文字、カタカナ、英数字、記号を含む18文字以内のファイル名を指定します。
なお、OSでは18文字のうち、はじめの8文字のみを識別しています。
全角文字は2文字として扱います。
拡張子の指定は次のように指定します。

.c
Cのソーステキストファイルであることを示す拡張子です。
この拡張子によってコンパイルします。

```
B>cc prog.c ↵
```

.s
アセンブラのソーステキストファイルであることを示す拡張子です。
この拡張子をつけた場合、Cコンパイラは起動されず、すぐにアセンブラが起動されます。
また、コマンドライン上に、.cをもったCのファイルと、.sをもったアセンブラのファイルを同時に指定した場合は、.cのファイルを最初にコンパイルして、次に.sのファイルをアセンブルします。

```
B>cc prog.c prog2.s ↵
```

. b (. bas)

X-BASICのソーステキストファイルであることを示す拡張子です。

CCコマンドラインには、X-BASICのソーステキストファイルを指定することができます。

これは「第5章 XBAStoC BASICプログラムコンバータ」で詳しく説明しますが、CCコマンドライン上に. bや. basの拡張子をもったファイルを指定すると、コンパイラは、それがX-BASICのソーステキストファイルであることを認識して、X-BASICのプログラムをC言語のプログラムに変換します。

後は、. cの拡張子で説明した通りに動作します。

```
B>cc bprog.b ⏎
```

. o

オブジェクトファイルであることを示す拡張子です。

CCのコマンドラインでは、ソーステキストファイルとともに、コンパイルおよび、アセンブル済みのオブジェクトファイルを指定することができます。

また、単独で、この拡張子のついたオブジェクトファイルを指定した場合は、すぐにリンカが起動されます。

```
B>cc prog.c exs.o ⏎
B>cc progl.o ⏎
```

予約ファイル名

次に示す予約ファイル名を使ってはいけません。

AUX, CON, PRN, LPT, PCM,
CLOCK, NUL, OPM

その他、デバイスドライバで、特殊に使用されているファイル名も使ってはいけません。

2.2 スイッチによる制御

XCコンパイラは、コンパイラの制御に関する多くのスイッチが用意されています。
スイッチの指定は‘／’, ‘—’で始まり、1文字以上の英字からなります。
また、‘／’か‘—’は、使用するかたが自由に選ぶことができますが、原則として、‘／’を使用します。

なお、これからスイッチの説明をするうえで**スイッチ記号の大文字と小文字は区別してください。**

＊本節ではスイッチの表記はすべて‘／’により表記します。

2.2.1 スイッチ指定の規則

原則としてスイッチの指定は、ファイル名の指定より前に記述します。
コマンドラインは、以下のようになります。

```
CC ／＜スイッチ記号＞［＜パラメータ＞］．．．．＜ファイル名＞．．．．
  |                                  |     |            |
  |__________________________________|_____|____________|____ 空白
```

上記のように、各スイッチと＜ファイル名＞とは、空白で分離しなければなりません。
ただし、＜スイッチ記号＞とその＜パラメータ＞の間には空白を入れてはいけません。
もし、空白を入れてしまうと、コンパイラは、＜パラメータ＞を次のスイッチ、またはファイル名として認識してしまうからです。

2.2.2 スイッチの分類

XC コンパイラで使用できるスイッチは、すべて CC. X を実行するときに指定するスイッチです。

CC. X は指定されたスイッチによりコンパイルの実行を制御し、更に、CC. X 自身が起動するリンカ（LK. X）の制御も行います。

XC コンパイラで指定するスイッチは、次のように 6 つに分類できます。

1. 最適化に関するスイッチ
2. プリプロセッサに関するスイッチ
3. 出力ファイルに関するスイッチ
4. 言語に関するスイッチ
5. コード生成に関するスイッチ
6. その他の機能のスイッチ

2.2.3 スイッチの機能

本項では、前項で説明したスイッチの分類ごとに機能を説明します。

/O

最適化に関するスイッチ

機　能	プログラムの最適化
解　説	プログラムを最適化します。
省　略	最適化は行われませんが、その分コンパイルにかかる時間は短くなります。

例

```
B>cc  /O func.c ⏎
```

／C

プリプロセッサに関するスイッチ

機　　能　コメント行の保存

解　　説　このスイッチを指定すると、プリプロセッサはテキストファイルの注釈を保存します。
このスイッチは／P、／Vとともに指定される場合に有効です。

省　　略　プリプロセッサはテキストファイルの注釈をすべて削除します

例

```
B>cc  /C  /P  func.c ⏎
```

／D<識別子> [=定数]

プリプロセッサに関するスイッチ

機　能　マクロ定義

解　説　このスイッチはプリプロセッサ命令である #define と同様にマクロを定義します。

このスイッチによるマクロ定義は、最大20個までです。

21個目からは無視します。

省　略　ソースプログラム中に記述されている #define 命令で定義されているマクロだけが有効になります。

例

```
B>cc  /Dval=200  func.c ⏎
B>cc  /DXC  /DMPU=68K  /DKANJI sample.c ⏎
```

／I<パス名>

プリプロセッサに関するスイッチ

機　能　インクルードパスの指定

解　説　このスイッチにより、インクルードファイルのパスを指定または変更できます。
独自のインクルードファイルを作成した場合などに使用できます。
このスイッチによるインクルードパスの指定は、最大8個までです。
9個目からは無視します。

省　略　環境変数includeに設定されているパスを検索します。

例

```
B>cc  /Ib:¥myinc  /Ic:¥include func.c ↵
```

この例では、b:¥myinc、c:¥includeよりインクルードファイルを検索します。

／P

プリプロセッサに関するスイッチ

機　能　プリプロセッサの出力をファイルへ出力

解　説　プリプロセッサの出力リストをファイルに出力します。
ファイル名はソースファイルと同一名で拡張子は.pになります。
この出力リストは注釈が削除されていますので、必要な場合は／Cスイッチも指定して、注釈を残してください。
このスイッチを指定した場合は、プリプロセッサ処理のみで終了します。

省　略　プリプロセッサの出力はテンポラリファイルに出力され、コンパイルの処理は続行されます。
テンポラリファイルはコンパイル終了時に削除されます。

例

```
B>cc  /P  /C func.c⏎
```

この例では、プリプロセッサの出力リストをfunc.pに出力します。
また、注釈は保存されます。

／U<識別子>

プリプロセッサに関するスイッチ

機　能

マクロ定義の取り消し

解　説

このスイッチは、プリプロセッサ命令である #undef と同様に、<識別子>で指定されたマクロ定義を取り消して、未定義とします。

プリプロセッサは、ここで指定された識別子がソースプログラム中に記述されていた場合、未定義のマクロとして取り扱います。

このスイッチで、最大20個までの識別子を未定義とすることができます。

21個目からは無視します。

／U と識別子の間にはスペースを入れてはいけません。

また、以下に示す定義済みマクロを取り消してはいけません。

__STDC__, __VERSION__, __LINE__, __FILE__, __TIME__, __DATE__

省　略

ソースプログラム中に記述されている #undef 命令によるマクロ定義の取り消しのみが有効となります。

例

```
B>cc  /Dval1=100  /Dval2=200  /Dval3=300  /Uval1 func.c ↵
```

この例では、識別子 val1 のみが未定義となります。

/V

機　　能　プリプロセッサの出力を標準出力へ出力

解　　説　このスイッチを指定することにより、プリプロセッサは出力リストを標準出力に出力します。

この出力リストは注釈が削除されていますので、必要な場合は／Cスイッチも指定して注釈を残してください。

このスイッチを指定した場合は、プリプロセッサ処理のみで終了します。

省　　略　プリプロセッサの出力は、テンポラリファイルに出力され、コンパイルの処理は続行されます。

テンポラリファイルは、コンパイル終了時に削除されます。

例

```
B>cc  /V  func.c ↵
```

/Fc

出力ファイルに関するスイッチ

機能　リンクの抑制

解説　このスイッチの指定により、CC.X はリンカを起動せずに、オブジェクトファイルのみを出力して終了します。
出力されるオブジェクトファイルは、ソースファイル名の拡張子を.o に置き換えたファイル名になります。

省略　CC.X はリンクまで実行し、実行可能ファイルを作成します。

例

```
B>cc  /Fc func.c ⏎
```

／Fd<ファイル名>

出力ファイルに関するスイッチ

機　　能	アセンブラソースファイル（.s）の出力ファイル名の指定
解　　説	このスイッチとそれに続くファイル名を指定することにより、アセンブラソースファイルを指定されたファイル名で出力します。 このスイッチは、／Fs スイッチとともに指定した場合に有効です。
省　　略	／Fs スイッチを指定した場合、カレントディレクトリ上にソースファイル名と同一名で拡張子が、.s のアセンブラソースファイルを出力します。

例

```
B>cc  ／Fs  ／Fdc:¥usrl¥sample¥sample.s sample.c ⏎
```

この例では、ソースファイル sample.c をコンパイルして作成されたアセンブラソースプログラムを、c:¥usrl¥sample¥sample.s というファイルに出力します。

/Fo<ファイル名>

出力ファイルに関するスイッチ

機　能　オブジェクトファイル（.o）の出力ファイル名の指定

解　説　このスイッチとそれに続くファイル名を指定することにより、オブジェクトファイルを指定されたファイル名で出力します。

省　略　カレントディレクトリ上にオブジェクトファイルを出力します。

例

```
B>cc  /Foc:¥usr1¥objs¥sample.o sample.c ↵
```

この例では、ソースファイル sample.c をコンパイルして作成されたオブジェクトを、c:¥usr1¥objs¥sample.o というファイルに出力します。

／Fs

出力ファイルに関するスイッチ

機　能　アセンブラソースファイル（.s）のみの作成

解　説　コンパイルを実行し、アセンブラソースファイル（.s）が作成されます。
アセンブル・リンクは実行されません。

省　略　コンパイル、アセンブル、リンクのすべてが行われます。
アセンブラソースファイルは作成されません。

例

```
B>cc  ／Fs func.c ⏎
```

／Fx<ファイル名>

出力ファイルに関するスイッチ

機　能　実行可能ファイル（.X）の出力ファイル名の指定

解　説　このスイッチとそれに続くファイル名を指定することにより実行可能ファイルを指定されたファイル名で出力します。

省　略　CC.X 起動時のコマンドライン上に最初に指定されたソースファイル名に拡張子 X がついたファイル名として出力します。

例

```
B>cc　／Fxc:¥bin¥sample.x sample.c ↵
```

この例では、ソースファイル sample.c をコンパイルして作成された実行可能プログラムを、c:¥bin¥sample.x というファイルに出力します。

/Nf

機　　能　浮動小数点演算ライブラリの変更

解　　説　浮動小数点演算ライブラリを FLOATEML. L に変更します。
デフォルトの FLOATFNC. L を使用した場合は、FLOAT?. X（FLOAT2. X など）が組み込まれていなければ、実行できない実行ファイルが作成されます。
FLOATEML. L を使用した場合は、FLOAT?. X が組み込まれていなくても実行可能な実行ファイルが作成されますが、FLOAT?. X を使用する実行ファイルに比べてファイルサイズが大きくなります。

省　　略　浮動小数点演算ライブラリは FLOATFNC. L を使用します。

例

```
B>cc　/Nf sample.c ⏎
```

／NI＜パス名＞

言語に関するスイッチ

機　　能　ライブラリパスの指定

解　　説　スイッチとそれに続く＜パス名＞を指定することにより、リンク時に指定したライブラリをリンカが認識します。
ユーザーが独自で作成したライブラリを指定する時に使用します。

省　　略　環境変数 lib に設定されているパスを検索します。

例

```
B>cc  /Nla:¥usr¥lib¥usrlib.l  func.c ↵
```

／Ns

機　　能　ソースコードデバッガ（SCD. X）用コードの生成

解　　説　ソースコードデバッガ SCD. X による、ソースプログラムレベルでのデバッグを可能にするために、実行可能ファイルにデバッグ情報を付加して出力します。
このスイッチを指定する場合は、以下のスイッチを同時に指定してはいけません。

　／O　／Gh　／Gs

省　　略　デバッグ情報は付加されません。

例

```
B>cc  /Ns  func.c ↵
```

/Gc

コード生成に関するスイッチ

機　能　スタック検査の指定

解　説　このスイッチが指定されると、プログラム実行時にスタック領域に十分なスペースがあるかどうかをチェックするコードを実行可能ファイルに付加します。
十分なスペースがない場合には、その旨のメッセージを表示します。

省　略　スタックチェックを行いません。
スタックチェックを行わないと、スタックがオーバーフローを起こす可能性がありますが、実行速度は、スタックチェックを行うコードを含まない分高速です。

例

```
B>cc  /Gc func.c ↵
```

／Gh<サイズ>

コード生成に関するスイッチ

機　能　ヒープサイズの設定

解　説　実行時のヒープサイズを1バイトもしくは1Kバイト単位で指定します。
malloc関数などで、大きな領域が必要なときにこのサイズを大きくとる必要があります。
ヒープサイズの最小値は8Kバイト、最大値はメモリのサイズによって変わります。

省　略　デフォルトのヒープサイズは64Kバイトです。

例

```
B>cc  /Gh128k func.c ↵
```

この例では、ヒープサイズを128Kバイトに設定しています。

／Gs<サイズ>

コード生成に関するスイッチ

機　能　スタックサイズの設定

解　説　実行時のスタックサイズを1バイトもしくは1Kバイト単位で指定します。
プログラムで、リカーシブルコール（再帰呼び出し）を多用している場合などでは、実行時のスタックを大量に消費するため、このスイッチによりスタックサイズを大きくとっておく必要があります。
スタックサイズの最小値は4Kバイト、最大値はメモリサイズによって変わります。

省　略　デフォルトのスタックサイズは64Kバイトです。

例

```
B>cc  /Gs90k func.c ⏎
```

この例では、スタックサイズを90Kバイトに設定しています。

/A<レベル>

その他の機能のスイッチ

機　　能　警告レベルの指定

解　　説　ワーニングエラーの構文チェックレベルを指定します。
構文チェックレベルの範囲は、1~3です。
各レベルの内容は次の通りです。

レベル1……通常の構文チェックレベルで、ワーニングエラーを出力します。
レベル2……明示的な型変換が発生したときに、ワーニングエラーを出力します。
レベル3……暗黙的な型変換が発生したときに、ワーニングエラーを出力します。

省　　略　構文チェックレベルはレベル1です。

例

```
B>cc  /A3 func.c ⏎
```

／B

その他と機能のスイッチ

機　能　C言語ソースファイル（.C）のみの作成

解　説　このスイッチを指定した場合、コマンドラインにはX-BASICのソースファイルを指定します。
その場合、C言語のソースファイルを出力してコンパイルを終了します。
C言語のソースファイルは、コマンドラインで指定したX-BASICのソースファイル名の拡張子を.cで置き換えたファイル名で出力します。

省　略　C言語のソースファイルを出力して、コンパイルを最後まで実行します。

例

```
B>cc　／B func.bas ⏎
```

/E<長さ>

機　　能	識別子の最大長の指定
解　　説	C 言語の識別子の長さを指定します。 長さは、8～32 の範囲で指定します。
省　　略	識別子の最大長は 32 です。

例

```
B>cc  /E20 func.c⏎
```

／J

その他の機能のスイッチ

機　能　char 型を unsigned char 型とする

解　説　char 型は、デフォルトでは符号付きとなっていますが、このスイッチを指定するとデフォルトを符号なし（unsigned）に変更します。
したがって、char の値を int に拡張した場合、このスイッチを付けないと符号拡張されますが、付けると符号拡張されません。

省　略　char 型を符号付きとします。

例

```
B>cc　／J　func.c ↵
```

/Q

機　　能　同一コード文字例の圧縮

解　　説　プログラム中に、同じ内容の文字例が初期値として複数個使用されていたとき、その文字例を１つだけに圧縮して、コンパイルします。

省　　略　圧縮は行われません。

例

```
B>cc　／Q　func.c ⏎
```

／T<パス名>

その他の機能のスイッチ

機　　能　作業パスの指定

解　　説　コンパイル時に使用する作業用ディレクトリを指定します。

省　　略　環境変数 tempで指定されたパスとなります。
環境が指定されていない場合は、カレントディレクトリとなります。

例

```
B>cc  /Tc:¥tmp func.c ⏎
```

／Y

機　能	DOS／IOCS ライブラリの使用フラグ
解　説	このスイッチを指定することによりDOS／IOCSのライブラリ（DOSLIB.L, IOCSLIB.L）をリンカが認識します。
省　略	リンク時にDOSLIB.LとIOCSLIB.Lを使用しません。

例

```
B>cc　／Y　func.c ⏎
```

／W

その他の機能のスイッチ

機　能　BASIC ライブラリの使用フラグ

解　説　このスイッチを指定することにより BASIC ライブラリ（BASLIB.L）をリンカが認識します。

省　略　BASLIB.L をリンク時に使用しません。

例

```
B>cc　／W func.c ↵
```

／G0<シンボル数>

その他の機能のスイッチ

機　　能	シンボルの最大個数の指定（パーザ）
解　　説	パーザが処理するシンボルの、最大個数を指定できます。 シンボルの最大個数の範囲は、以下に示す通りです。 400 ≦ シンボル数 ≦ 65535
省　　略	デフォルトのシンボル数は、3000個です。

例

```
B>cc　／G050000 test.c⏎
```

／Ga<シンボル数>

その他の機能のスイッチ

機　　能　シンボルの最大個数の指定（AS部）

解　　説　ASが処理するシンボルの最大個数を指定できます。
シンボルの最大個数の範囲は、以下に示す通りです。
270 ≦ シンボル数 ≦ 32768

省　　略　デフォルトのシンボル数は、2000個です。

例

```
B>cc  ／Ga5000 test.c↵
```

／Gb<バッファ数>

機　　能　ハッシュバッファの最大個数の指定（BC部）

解　　説　BCがシンボルを処理するために使用するハッシュバッファ（1997個のエントリ）の、最大個数を指定できます。
ハッシュバッファの最大個数の範囲は、以下に示す通りです。
1 ≦ バッファ数 ≦ 32
シンボルの最大個数は、1997×バッファ数となります。

省　　略　デフォルトのバッファ数は、1個です。

例

```
B>cc  /Gb2 test.bas ⏎
```

／Gp<シンボル数>

その他の機能のスイッチ

機　能　シンボルの最大個数の指定（プリプロセッサ部）

解　説　プリプロセッサが処理するシンボルの最大個数を指定できます。
シンボルの最大個数の範囲は、以下に示す通りです。
1000 ≦ シンボル数 ≦ 65535

省　略　デフォルトのシンボル数は、2000 個です。

例

```
B>cc  ／Gp5000 test.c ↵
```

／Na

機　　能　32K バイト以上の auto の変数の使用許可

解　　説　32K バイト以上の auto 変数を使用できます。
ただし、32K バイト以上の auto 変数を使用したときは、警告メッセージが表示されます。

省　　略　このスイッチを省略したときは、32K バイト以上の auto 変数を使用できません。
32Kバイト以上のauto変数を使用したときは、エラーメッセージが表示されます。

例

```
B>cc  ／Na test.c↵
```

/X

その他の機能のスイッチ

機　　能　プリプロセッサの定義済み識別子__STDC__を定義しません

解　　説　プリプロセッサの定義済み識別子__STDC__を定義しません。
その結果、標準ライブラリ関数はプロトタイプ宣言されません。
標準ライブラリを使用するとき、引数の型チェックを行いません。

省　　略　プリプロセッサの定義済み識別子__STDC__は1に定義されます。
その結果、標準ライブラリ関数はプロトタイプ宣言されます。
標準ライブラリ関数を使用するときは、引数の型チェックを行います。

例

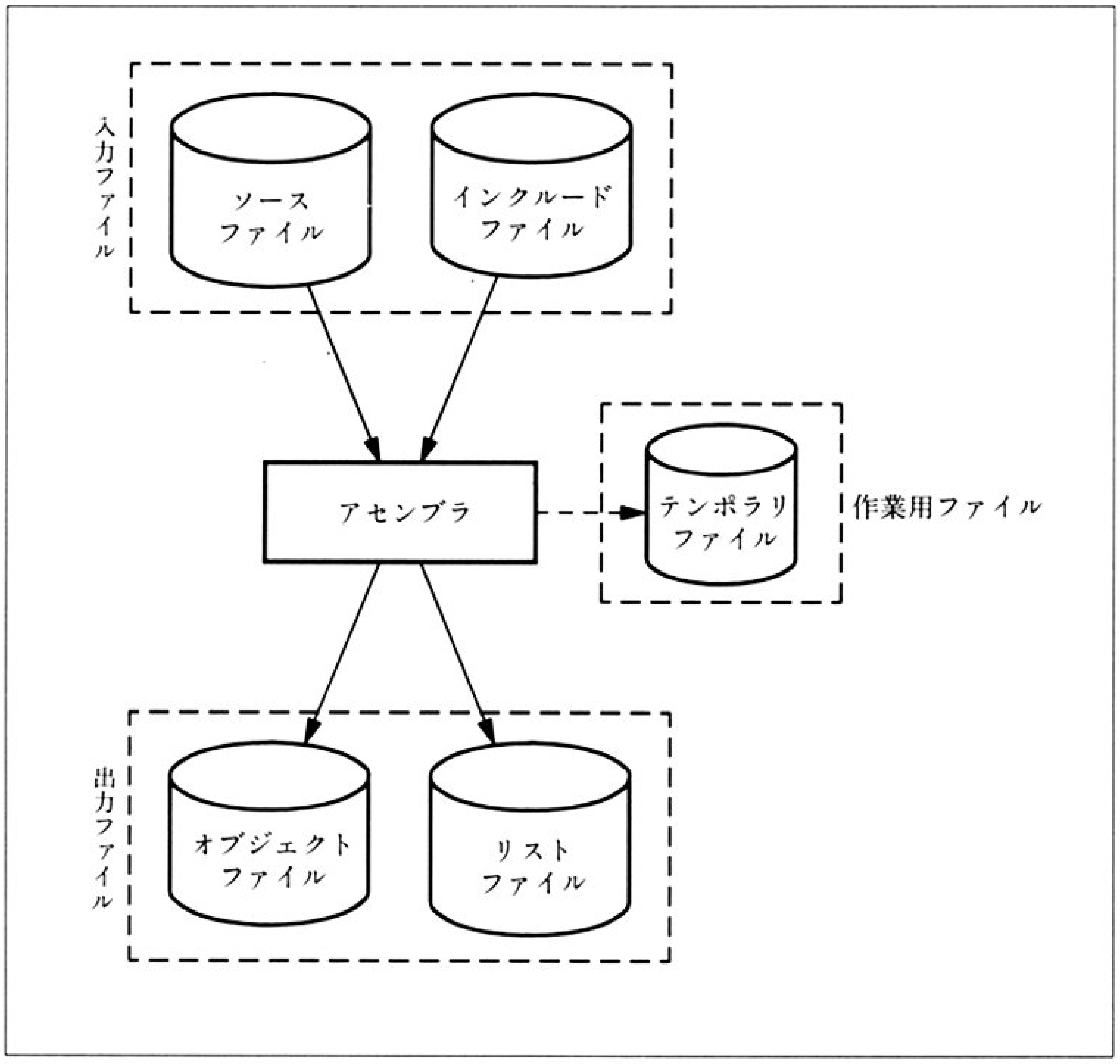
```
B>cc  /X test.c ↵
```

2.3 アセンブル

本節では、XC コンパイラのアセンブラ（XAssembler）について概要を説明します。
詳しい説明は、「アセンブラマニュアル」を参照してください。

2.3.1 アセンブラで使う入出力ファイル

アセンブラを実行するうえでのファイル構成と内容について説明します。



2.3　アセンブル

(1)入力ファイル

入力ファイルには、ソースファイルとインクルードファイルの2種類があります。

ソースファイル

アセンブリ言語のニーモニックで記述されたプログラムが格納されているファイルです。

インクルードファイル

これもソースファイルですが、各プログラム間で共通に使用するシンボルなどを定義するときに使用します。
インクルードファイルを使用するときは、アセンブラのソースプログラム中に .include 疑似命令を記述します。
この指定によりインクルードファイルの内容がソースプログラム中に挿入されるので、プログラムのコーディングの手間が省けるとともに、プログラム開発の生産性が向上します。

(2)テンポラリファイル

これは、アセンブル時の作業用ファイルです。
アセンブル時にはテンポラリファイルが常に作成されますので、アセンブルが終了するまでフロッピーディスクをとりはずしてはいけません。
アセンブルが終了すると、テンポラリファイルは自動的に削除されます。
なお、テンポラリファイルのパス名は環境変数 temp より参照されます。
ただし、／t スイッチで指定された場合は、こちらの指定が優先されます。

(3)出力ファイル

出力ファイルには、オブジェクトファイルとリストファイルの2種類があります。

オブジェクトファイル

アセンブル処理によって作成されたオブジェクトプログラムを格納するファイルです。
オブジェクトファイル名は、／o スイッチで指定します。
指定がなければ、アセンブラはソースファイルの拡張子. s を. o で置き換えたファイルを自動的に作成します。

リストファイル

アセンブルリストを格納するファイルです。

リストファイル名は、／p スイッチで指定します。

／p スイッチのみを指定してリストファイル名を指定しないと、アセンブラはソースファイル名の拡張子 s を prn で置き換えたファイルに自動的に作成します。

2.3.2 アセンブラの使用書式

アセンブラは次の書式で使用します。

```
as [＜スイッチ＞] ＜ソースファイル名＞
```

2.3.3 アセンブラのスイッチ

アセンブラ実行時に、アセンブル時の動作を制御したり、アセンブラに必要な情報を与えるために、コマンドライン上に指定する英字1文字をスイッチといいます。

```
B>as /t¥tmp /oobjout source ↵
      |        |
      +--------+-- スイッチ
```

XAssembler には以下のようなスイッチがあります。

スイッチ名	意　味
／8	シンボルの識別長を8バイトにする
／a	絶対ショートアドレス形式対応モード
／d	すべてのシンボルを外部定義する
／i	インクルードのパス指定
／m	最大シンボル数
／n	最適化の禁止
／o	オブジェクトファイル名
／p	リストファイル
／s	シンボルの定義
／t	テンポラリーパス指定
／u	未定義シンボルを外部定義にする
／w	ワーニングエラーの出力禁止
／x	シンボルの出力

2.4 リンク

本節では、リンカ（XLinker）についての概要を説明します。
詳しい説明は、「アセンブラマニュアル」を参照してください。
リンカとは、アセンブルしたオブジェクトをリンクして実行可能プログラムを作成するツールのことをいいます。
リンカの機能は、次の通りです。

(1)別々にアセンブルしたオブジェクトを結合します。
(2)未定義の外部参照をライブラリファイルから探し、外部参照を解決します。
(3)外部参照の解決やエラーメッセージを示すリストを作成します。

高級言語であるCなどのコンパイラで作成したアセンブラソースや、初めからアセンブラで作成したソースは、いったんアセンブルされると、同一に扱うことのできるオブジェクトになります。
通常、開発はプログラムを複数のモジュールに分割して作成・アセンブル、その後にまとめ上げるという手法をとります。
たとえば、C言語でプログラムの大半、特にアルゴリズムの中心部分を作成し、細かな制御が必要なI/Oに関連する部分をアセンブラで作成し、これらを組み合わせるなどです。
このように、オブジェクトをまとめて1つの実行可能プログラムを作成することがリンカの働きです。

また、C言語などでは、関数の多くがオブジェクトの形式でライブラリとしてサポートされているので、そのためにもリンカが必要です。

リンカは、分割されている複数のオブジェクトをまとめ上げます。
このとき、別のオブジェクトの一部を利用するルーチンを参照したり、ライブラリ関数を参照するときには、まず、オブジェクトを配置して1つにまとめ上げ、次に外部の参照の関係を正しくします。

2.4.1 リンカで使う入出力ファイル

リンカは、入力、出力およびテンポラリファイルを使用します。

(1)入力ファイル

種　類	拡張子	作成経緯
オブジェクト	o	コンパイラまたはアセンブラ
アーカイブ	a	アーカイバ
ライブラリ	l	ライブラリアン

(2)出力ファイル

種　類	拡張子	使用目的
実行ファイル	x	プログラム実行
マップファイル	map	参考資料

(3)テンポラリファイル

リンカは、実行可能ファイルの作成に使用するデータの保持に記憶領域を使用可能なかぎり使います。

ここで処理するオブジェクトが大きい場合、リンカはさらに記憶領域を必要とします。

そこで、ディスクにテンポラリファイルが自動的に作成されます。

このようにリンカはテンポラリファイルを作成するので、この処理が終了するまでフロッピーディスクをとりはずしてはいけません。

リンカが終了するとテンポラリファイルは自動的に消去されます。

このため通常の使用では、このファイルは必ず作成されるので、あらかじめ十分使用可能なディスク容量をディスクにとっておかなければいけません。

そこで、テンポラリファイルを作成するためのパスを指定しておくことができます。

テンポラリファイルの指定には、／t スイッチを使用します。

```
B＞lk　／tb：temp pp⏎
　　　テンポラリファイル指定
```

／t スイッチによってテンポラリファイルを指定しないときは、環境変数 temp を参照します。
temp の指定があっても、／t スイッチによる指定があれば、スイッチの指定を優先します。

2.4.2 リンカの使用書式

リンカは、次の書式で使用します。

```
lk ［＜スイッチ＞］＜オブジェクトファイル名＞［＜オブジェクトファイル名＞…］
```

オブジェクトファイル名は、複数個指定できます。
また、オブジェクトファイル名の他に、オブジェクトをまとめたライブラリファイル名も指定できます。
ライブラリファイルを指定すると、リンカは自動的にその中に含まれる必要なオブジェクトを探します。

2.4.3 リンカのスイッチ

リンカのスイッチは、リンクエディット時の動作を制御したり、リンクエディットに必要な情報を与えるために使用します。
リンカには、以下のようなスイッチがあります。

スイッチ名	意　味
／b	ベースアドレス指定
／i	インダイレクトファイルの指定
／m	最大シンボル数
／o	オブジェクトファイル名
／p	マップファイルの出力
／t	テンポラリパス指定
／v	バーボーズモード
／x	シンボルテーブルの出力禁止

アーカイブファイル（拡張子がaのファイル）を高速にリンクしたい場合は、次のようにファイルを変換してください。
（注）通常はライブラリファイル（拡張子がlのファイル）をご使用ください。

```
B>lib  user.l  user.a
```

第3章

プログラムの実行

実行方法

コマンドライン

本章では、XC コンパイラによって作成された実行可能ファイルの実行に関する説明をします。

3.1 実行方法

XCのソースファイルは、コンパイルされて.xという拡張子のついた実行可能ファイルとして出力されます。
この実行可能ファイルは、プロンプトから拡張子.xをつけないで、入力することにより実行することができます。

```
B＞sample⏎
```

上記の例は、sample.xという実行可能ファイルを実行したところです。
実行可能ファイルは、X68000のOS、Human68kの実行可能ファイルとして動作します。

3.2 コマンドライン

前章では、CC. Xのコマンドラインについて説明しました。
本節では、ユーザーがC言語によって作成した実行可能ファイルのコマンドラインについて説明します。

3.2.1 データをプログラムへわたす方法

C言語で作成するプログラムは、実行するときに同時にデータを引きわたすことができます。
これは、コマンドライン上に実行可能ファイル名と同時に入力することにより実現できます。

```
B>sample  20  prog⏎
```

上記の例は、sampleというプログラムに、"20"と"prog"というデータをわたし、実行している例です。
各データを入力するときは、1つ以上の空白かTABによって区切らなければなりません。
データ中に空白を入力する場合は、クォーテーションマーク（"）により、区切る必要があります。

```
B>sample  100  "func 25"⏎
```

上記の例では、"100"というデータと"func 25"というデータをsampleにわたすことを意味しています。
このsampleというプログラムのmain関数は、ユーザーからデータを受け取る関数になっています。
このようなデータを受け取るmain関数は、次のように宣言する必要があり

ます。

```
main (int argc, char  *argv[] )
```

コマンドラインは、そのまま argv 文字列のデータとして main 関数に引きわたされます。
コマンドライン上のデータの数は、argc にわたされます。
次の例では、

```
B>sample  80  "X to Y"
```

"sample"というコマンド名は argv [0] に格納され、"80"は argv [1] に、"X to Y" は argv [2] に格納されることになります。
また、argc には、プログラム名とデータ数を含めた合計である 3 が格納されることになります。

第 4 章

アセンブラとの
インターフェイス

アセンブラインターフェイス
アセンブルプログラム例

本章では、C 言語とアセンブラをリンクするうえで必要になるインターフェイスについて説明します。
C 言語では実現しづらいプログラム、たとえばハードウェアを細かく操作しなければならないようなプログラムを作成する場合、実行スピードの面からも、アセンブラで作成した方が、より効果的です。
また単純に実行スピードだけを追求するようなプログラムにもいえることです。

XC コンパイラでは、このようなニーズを考慮して C 言語とアセンブラをリンクすることができます。

まず、本章では、アセンブラと XC のインターフェイスを一通り説明したあと、簡単な加算関数のいくつかの作成パターンを例にとって説明します。

4.1 アセンブラインターフェイス

本節では、C言語のプログラムとアセンブラのプログラムをリンクする場合、アセンブラプログラムでインターフェイスをどのようにとるかを説明します。
インターフェイスを大きく分けると、次のようになります。

(1)プログラムの入口と出口
(2)引数のアクセス
(3)戻り値の設定
(4)グローバル変数のアクセス
(5)名前の規則
(6)その他の注意

4.1.1 プログラムの入口と出口

入　口

一般的に、プログラムの入口では、次のような宣言や準備を行う必要があります。

①関数名を宣言する……xdef 命令使用
②引数を受け取る準備をする……offset や equ 命令で宣言する
③ auto 変数を確保する
例

```
link    a6,# AUTOSIZE
```

④レジスタを退避する
例

```
movem.l    d3-d7／a3-a5,-(sp)
```

出　口

プログラムの出口では、次のような手続きを踏まなければなりません。

①入口でセーブしたレジスタを復帰する

例

```
movem.l   (sp)+,d3-d7/a3-a5
```

② auto 変数を解放する

例

```
unlk   a6
```

③呼び出し元に戻る……rts 命令を使う

4.1.2 引数のアクセス

引数のアクセスでは、呼び出し元の引数がどこに格納されているかを知る必要があります。

一般的に引数は、スタックフレームというスタック領域の中に格納されています。

以下にスタックフレームの例を示します。

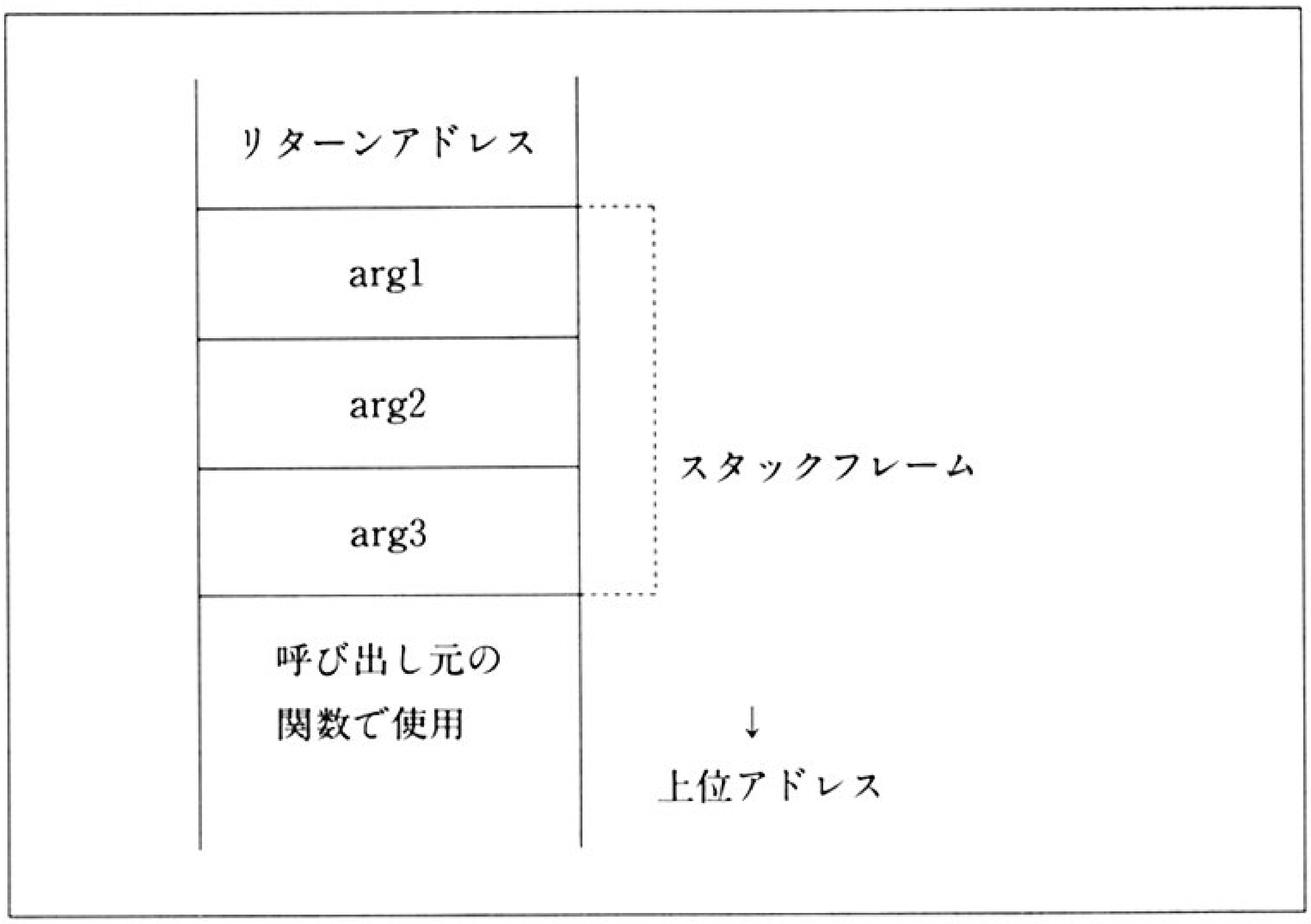
C 言語での関数仕様

```
sample (int arg1, int arg2, int arg3)
```

上記のような引数の場合、次のようなスタックフレームになります。

4.1　アセンブラインターフェイス

スタックフレーム



引数のとり込み方

上記のようなスタックフレームの場合、次のように引数をとり込みます。
また、スタックフレームに格納された引数の相対値は、次のように offset や equ などを使い、わかりやすくしておいたほうがよいでしょう。

例

```
                .offset     4
arg1            ds.l        1
arg2            ds.l        1
arg3            ds.l        1
                .text
savesize        equ         4＊（レジスタ退避数）
```

4.1 アセンブラインターフェイス

(1) int、short、char、ポインタ、float 型の引数の場合

①auto 変数使用時

②auto 変数を使用せず、かつレジスタの退避もしない場合

③auto 変数を使用せず、レジスタ退避をした場合

```
move.l    arg1(a6),d0  ………………………①
move.l    arg1(sp),d0…………………………②
move.l    arg1+savesize(sp),d0 …………③
```

(2) double 型の場合

```
movem.l   arg1(a6),d0-d1 ………………………①
movem.l   arg1(sp),d0-d1 ………………………②
movem.l   arg1+savesize(sp),d0-d1 ………③
```

(3) いくつかの引数を一度にとりこむ場合

```
movem.l   arg1(a6),d0-d1/a0 ……………………①
movem.l   arg1(sp),d0-d1/a0 ……………………②
movem.l   arg1+savesize(sp),d0-d1/a0 ……③
```

4.1.3 戻り値の設定

戻り値の設定方法は、設定する型によって異なってきます。
設定方法を次に示します。

型名	設定方法
char 型	d0 の下位 8 ビットに設定します
short 型	d0 の下位 16 ビットに設定します
int 型	d0 に設定します
float 型	d0 に設定します
double 型	d0、d1 に設定します
ポインタ	d0 に設定します

※ char, short, int 型に関しては unsigned タイプでも同じ設定方法です。

4.1.4 グローバル変数のアクセス

グローバル変数へのアクセスは、その変数が格納されているアドレスより直接読み込み／書き込みを行います。

グローバル変数の読み込み方

グローバル変数を読み込むときは、その変数が外部モジュールで宣言（領域確保）されている場合、次のような外部参照名の宣言を行う必要があります。

```
    . xref      _name
```

次に読み込み方の例を示します。

4.1 アセンブラインターフェイス

(1) char, short, int, float, double 型の変数の場合

これらの型は、次のように直接変数の値を読み込みます。

```
move. b    _name, d0 …………char 型変数
move. w    _name, d0……………short 型変数
move. l    _name, d0 ……………int 型変数
move. l    _name, d0 ……………float 型変数
movem. l   _name, d0－d1………double 型変数
```

(2)ポインタ変数の場合

ポインタ変数の値（ポインタ値）は、次のように読み込みます。

```
movea. l  _name, a0
```

ポインタ参照を行うには、上記のアドレスレジスタ（a0）を使用します。

例 1

```
move. b  (a0), d0
```

例 2

```
move. l  (a0), d0
```

例1と例2はそれぞれ char 型変数と int 型変数をポインタ参照した場合です。

(3)配列、構造体変数の場合

これらの型は、各変数の先頭のアドレスを次のように読み込みます。

```
lea    _name, a0
```

変数の各要素を参照するには、上記のアドレスレジスタ（a0）を使用します。

例 1

```
int       arry[8] ; ………………①

lea       _arry, a0         }
move. l   12(a0), d0        } ………②

move. l   12+_arry, d0 …………③
```

①のようなCの配列宣言の場合に、配列要素 arry[3] を参照するときの例が、②と③です。
このときのディスプレーメントは、4（int 型のバイト数）×3（添字）＝12 です。
②と③は同じ値を読み込みますので、場合により使い分けてください。

例2

```
struct    sample {
          int      x ;                 
          char     c ;                  ……④
} smpl [4] ;

lea       _smpl, a0                    ……⑤
move. b   22(a0), d0

move. b   22+_smpl, d0……………⑥
```

④のようなCの構造体を要素にもつ配列宣言の場合に、配列要素の構造体メンバsmpl [3].cを参照するときの例が、⑤と⑥です。

このときのディスプレーメントは、6（sizeof（struct sample）の値）×3（添字）+4（メンバcのオフセット）=22です。

⑤と⑥は同じ値を読み込みますので、場合により使い分けてください。

注：MPU68000では、奇数アドレスから始まるワード、ロングのデータアクセスはできませんでの（アドレスエラーが発生する）、構造体のメンバにchar型（バイトデータ）が含まれる場合は、自動的にワード境界の調整をコンパイラが行います。

したがって、そのような構造体をアセンブラでアクセスする場合は、注意が必要です。

4.1.5 名前の規則

アセンブラのラベル名のつけかたを説明します。
たとえば、Cでsmplという識別子はアセンブラでは_smplというように、ラベル名の前に“_”をつけます。
また、アセンブラの命令にある名前（moveやsp）は使えません。

4.1.6 確認

アセンブラプログラム内で壊してよいレジスタと、壊してはいけないレジスタがあります。
Cで使用するレジスタ変数は、d3／d4／d5／d6／d7、a3／a4／a5です。
また、システムではa6／a7を使用しているので、アセンブラプログラム内ではd0／d1／d2と、a0／a1／a2は壊してもかまいません。

4.2 アセンブラプログラム例

本節では、簡単な加算関数をアセンブラで作成するとどのようになるかを、いくつかのパターンを例にとって紹介します。
まず、Cからの呼び出し形式を示します。

```
int addsub (int src1, int src2, int * dst)
```

この関数の機能は、src1 と src2 を加算して dst に格納します。
また、この関数の戻り値として、src1－src2 の値を返すことになります。

例 1

a6／a7 以外のすべてのレジスタを使用する場合で、auto 変数を使用して、引数のとり込みをいろいろなところで行う場合。

```
        .xdef   _addsub         *  int addsub(src1,src2,dst)

        .offset 8               * arguments' offset

src1    ds.l    1               * int src1;
src2    ds.l    1               * int src2;
dst     ds.l    1               * int *dst;

AUTOSIZE        equ     -4
ret             equ     -4

        .text

_addsub                         * {
        link    a6,#AUTOSIZE    *       auto int        ret;
        movem.l d3-d7/a3-a5,-(sp)

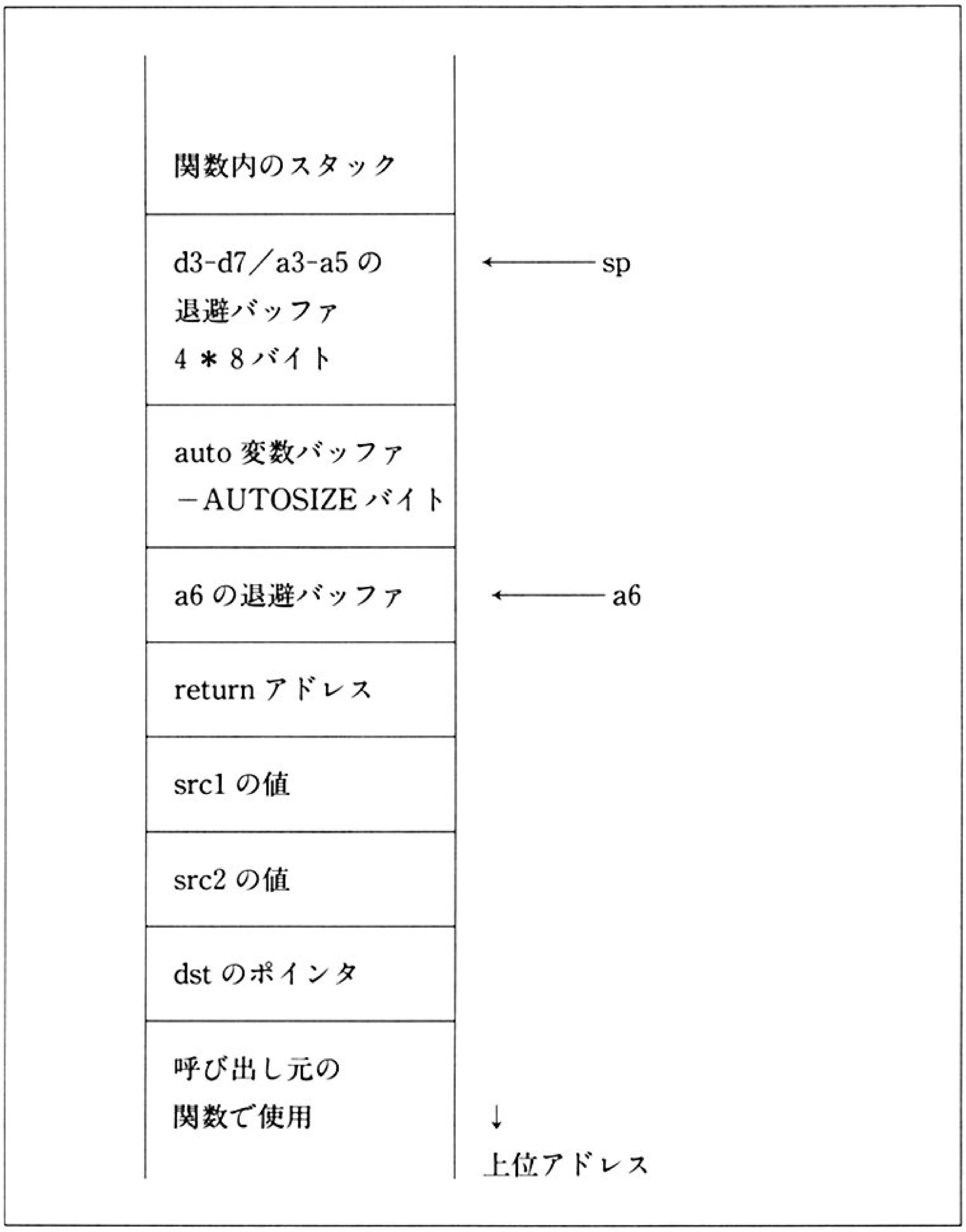
*       D0－D7／A0－A5のレジスタはすべて使用可

        move.l  src1(a6),d0
        movea.l dst(a6),a0
        move.l  d0,(a0)         *       *dst = src1;
        move.l  d0,ret(a6)      *       ret = src1;
        move.l  src2(a6),d0
        add.l   d0,(a0)         *       *dst += src2;
        sub.l   d0,ret(a6)      *       ret -= src2;
        move.l  ret(a6),d0      *       return(ret);

        movem.l (sp)+,d3-d7/a3-a5
        unlk    a6
        rts                     * }

        .end
```

スタックフレーム


関数内のスタック
d3-d7／a3-a5の
退避バッファ
4＊8バイト
sp
auto変数バッファ
－AUTOSIZEバイト
a6の退避バッファ
a6
returnアドレス
src1の値
src2の値
dstのポインタ
呼び出し元の
関数で使用
↓
上位アドレス

例２

d0-d2／a0-a2 のレジスタと auto 変数を使用して、引数のとり込みをいろいろなところで行う場合の例です。

C の関数は、この例のように作成されています。

```
        .xdef   _addsub         * int addsub(src1,src2,dst)

        .offset 8               * arguments' offset

src1    ds.l    1               * int src1;
src2    ds.l    1               * int src2;
dst     ds.l    1               * int *dst;

AUTOSIZE        equ     -4
ret             equ     -4

        .text

_addsub                         * {
        link    a6,#AUTOSIZE    *       auto int        ret;

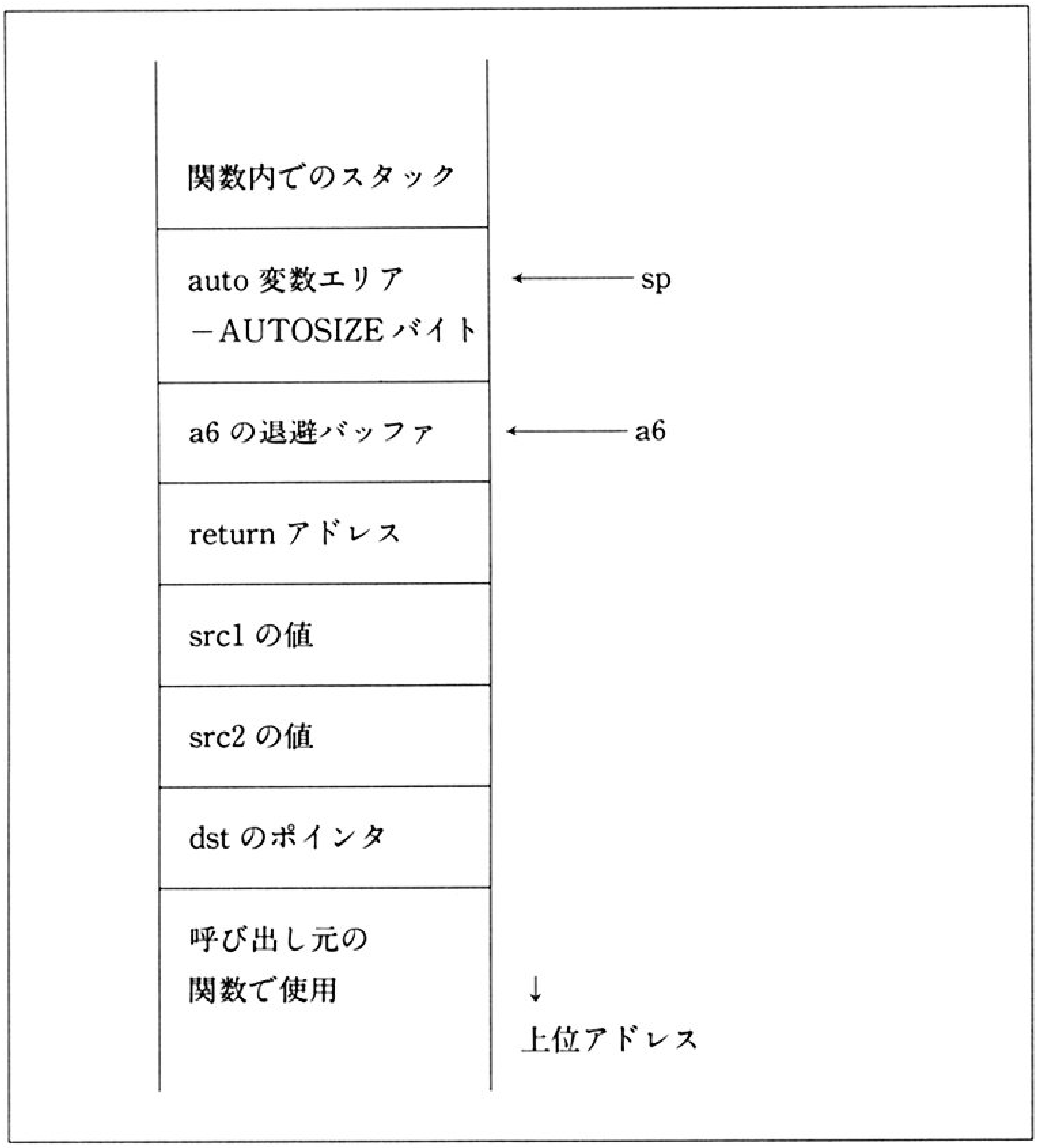
*       Ｄ０－Ｄ２／Ａ０－Ａ２のレジスタのみ使用可

        move.l  src1(a6),d0
        movea.l dst(a6),a0
        move.l  d0,(a0)         *       *dst = src1;
        move.l  d0,ret(a6)      *       ret = src1;
        move.l  src2(a6),d0
        add.l   d0,(a0)         *       *dst += src2;
        sub.l   d0,ret(a6)      *       ret -= src2;
        move.l  ret(a6),d0      *       return(ret);

        unlk    a6
        rts                     * }

        .end
```

スタックフレーム


関数内でのスタック
auto 変数エリア
－AUTOSIZE バイト
sp
a6 の退避バッファ
a6
return アドレス
src1 の値
src2 の値
dst のポインタ
呼び出し元の
関数で使用
↓
上位アドレス

例3

引数を入口ですぐとり込み、すべてのレジスタを使用したい場合の例です。

```
        .xdef   _addsub             * int addsub(src1,src2,dst)

        .offset 4                   * arguments' offset

src1    ds.l    1                   * int src1;
src2    ds.l    1                   * int src2;
dst     ds.l    1                   * int *dst;

        .text

savesize        equ     4*9

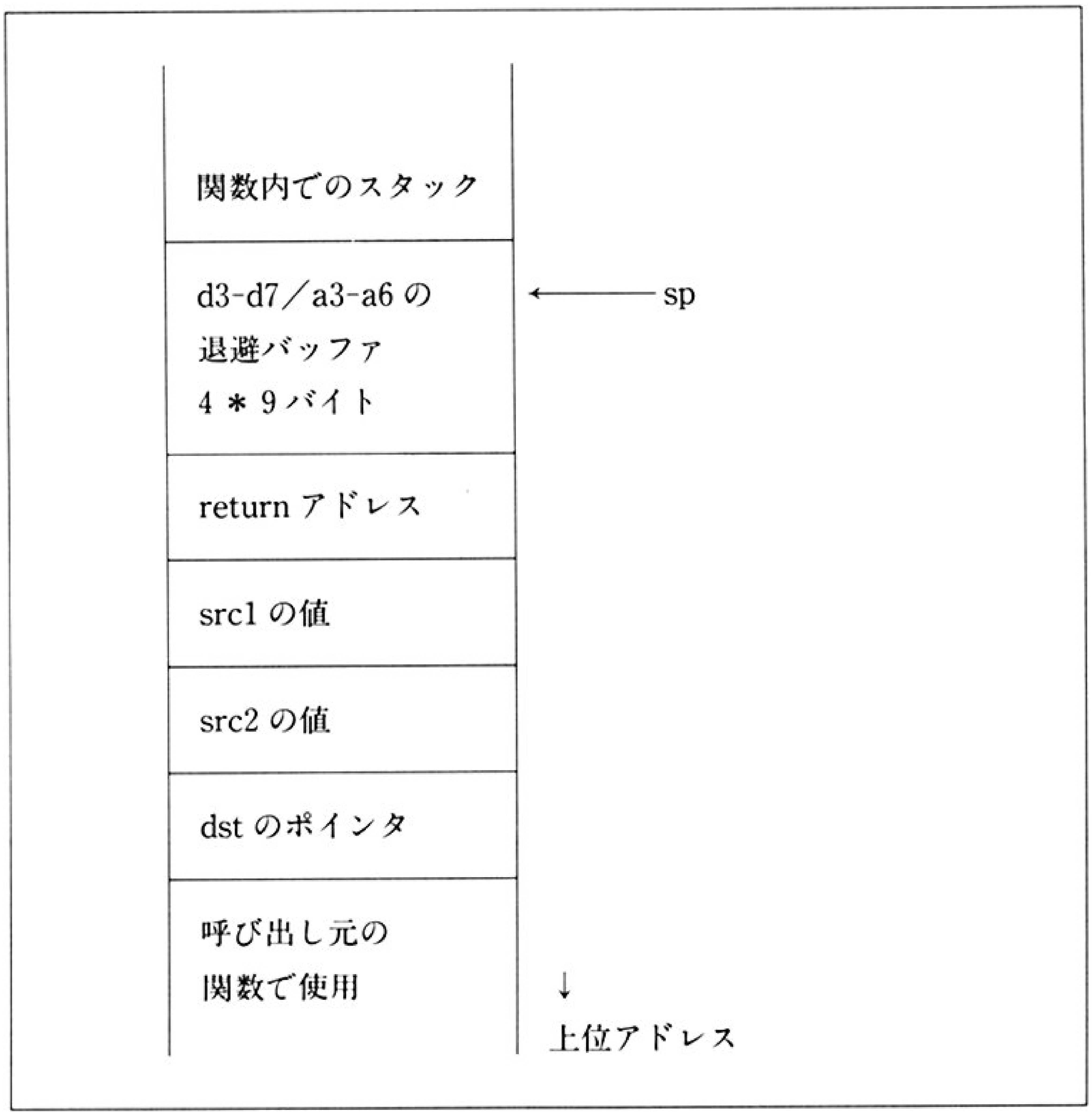
_addsub                             * {
        movem.l d3-d7/a3-a6,-(sp)

*       D0-D7/A0-A6のレジスタはすべて使用可

        movem.l src1+savesize(sp),d0-d1/a0
                                    * /* d0:src1 , d1:src2 , a0:dst */
        move.l  d0,(a0)             * *dst = src1;
        add.l   d1,(a0)             * *dst += src2;
        sub.l   d1,d0               * return(src1 - src2);

        movem.l (sp)+,d3-d7/a3-a6
        rts                         * }
        .end
```

スタックフレーム


関数内でのスタック
d3-d7／a3-a6の
退避バッファ
4＊9バイト
sp
returnアドレス
src1の値
src2の値
dstのポインタ
呼び出し元の
関数で使用
↓
上位アドレス

例 4

d0-d2／a0-a2 のレジスタのみで処理できる場合。

```
        .xdef   _addsub          * int addsub(src1,src2,dst)

        .offset 4                * arguments' offset

src1    ds.l    1                * int src1;
src2    ds.l    1                * int src2;
dst     ds.l    1                * int *dst;

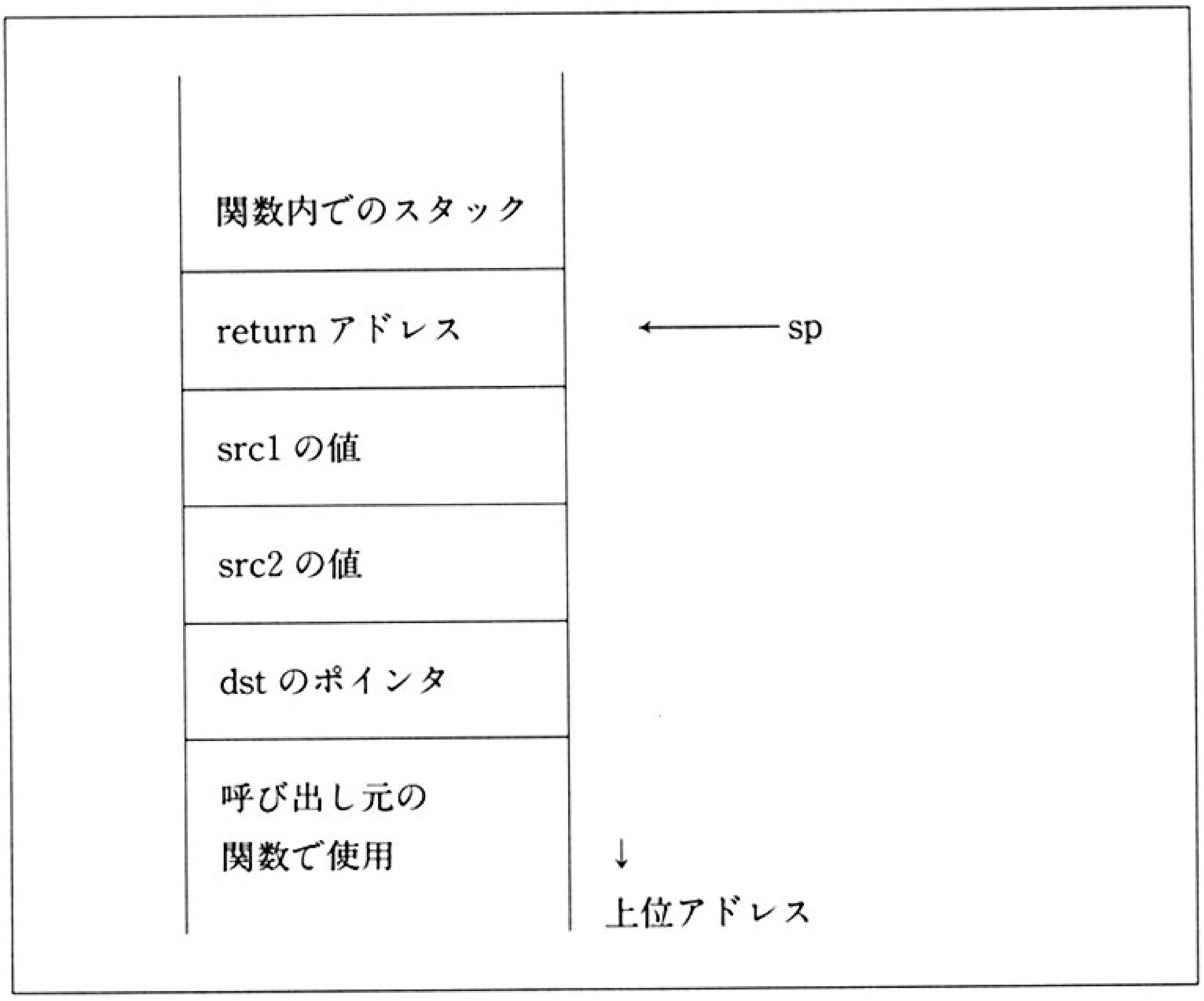
        .text

_addsub                          * {

*       Ｄ０－Ｄ２／Ａ０－Ａ２のレジスタのみ使用可

        movem.l src1(sp),d0-d1/a0
                                 * /* d0:src1 , d1:src2 , a0:dst */
        move.l  d0,(a0)          * *dst = src1;
        add.l   d1,(a0)          * *dst += src2;
        sub.l   d1,d0            * return(src1 - src2);
        rts                      * }
        .end
```

スタックフレーム


関数内でのスタック
returnアドレス
sp
src1の値
src2の値
dstのポインタ
呼び出し元の
関数で使用
↓
上位アドレス

第5章

XBAStoC BASIC プログラムコンバータ

BASTOCとは
プログラムの作成の流れ
変換方法
BASTOCでのBASICプログラムの書きかた
BASTOCのファイル構成

本章では、BASIC プログラムから C 言語のプログラムへ変換するコンバータ、XBAStoC(以下 BASTOC と呼びます)の説明をします。
通常の X-BASIC のプログラムをそのまま変換してもかまいませんが、変換後、C のコンパイルを行うため、X-BASIC のプログラムに多少の制限事項をもうけてあります。
BASTOC の利用を前提に X-BASIC のプログラムを作成する場合は、「5.4 BASTOC での BASIC プログラムの書きかた」をあらかじめお読みください。

5.1 BASTOCとは

BASICは、もともと初心者向けに作られた言語であり、その開発の容易さにより、古くから多くのユーザーに親しまれてきました。
X68000のX-BASICもHuman68k上で走るインタープリタBASICであり、従来のBASICをさらに強化し、関数定義やローカル変数の扱い、さらに、データ型のサポートといった、きめの細かい処理を可能にしています。
なお、X-BASICの詳しい説明は、“BASICマニュアル”を参照してください。
このX-BASICで記述したプログラムをC言語に変換するコマンドが、ここで説明するBASTOCです。
BASTOCは“BASic TO C”の大文字をとった略で、X-BASICのプログラムを組むことはできますが、C言語は初心者、というようなかたに最適なコマンドです。
また、X-BASICである程度デバック作業を行い、完成したあと一括してC言語に変換することも可能です。
以下にBASTOCの特長を示します。

BASTOCの特長

① XCコンパイラと同様に、コマンドラインにより実行することができます。

② BASICと同等の関数が用意されています。

5.2 プログラムの作成の流れ

本節では、BASTOC を使ってプログラムを作成する場合の流れを説明します。

BASTOC は X-BASIC から C 言語へのコンバータですので、初めに X-BASIC のプログラムを作成しなければなりません。

また、C 言語に変換しやすいようなプログラムの作成方法がありますが、本節では開発の流れを中心に説明しますので、X-BASIC のプログラムを作成するうえでの注意事項は、「5.4　BASTOC での BASIC プログラムの書きかた」で説明します。

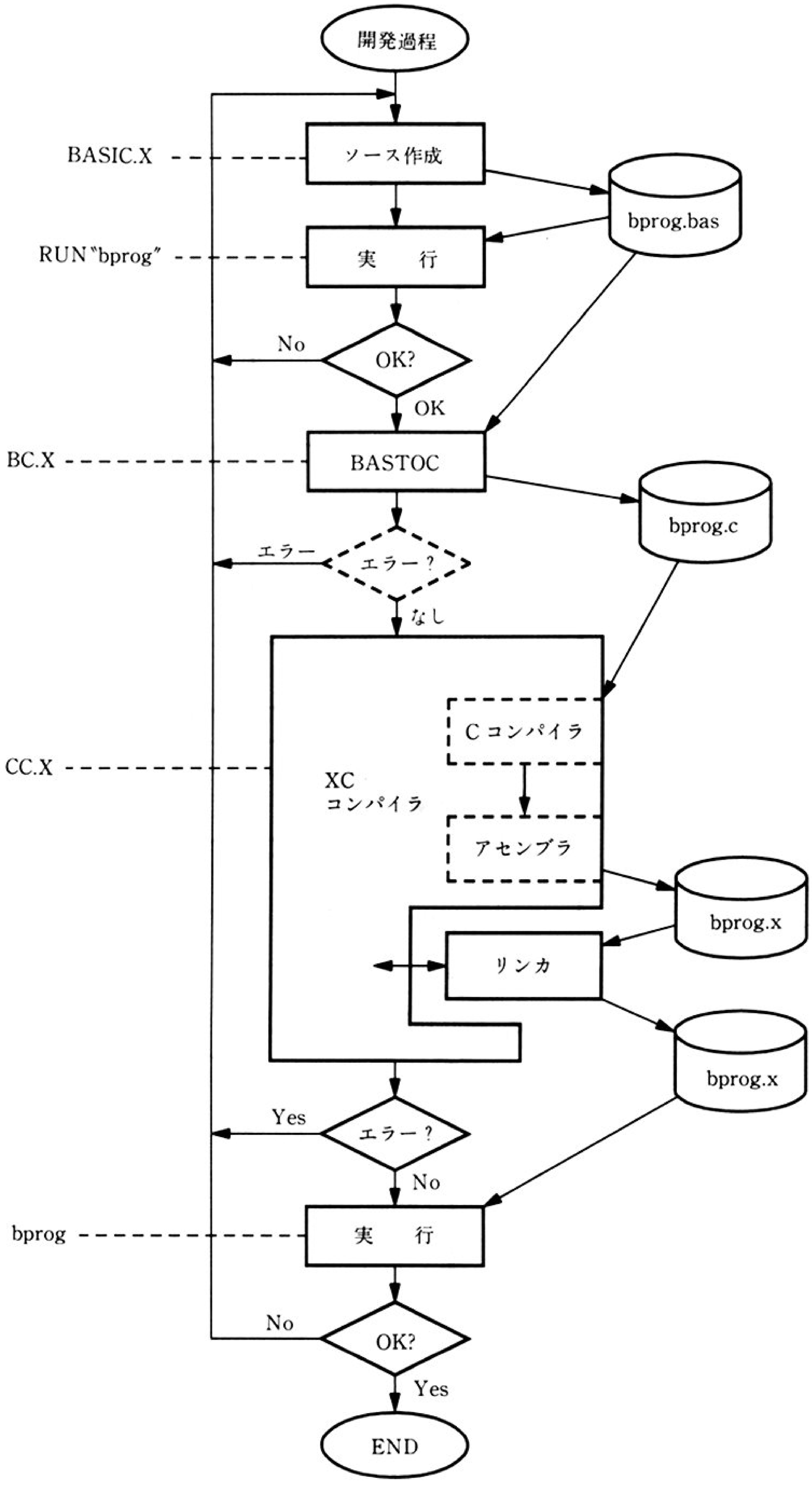
プログラム開発には、大きく分けて次の2通りがあります。

(1) X-BASIC で独自にデバッグ作業を行い、動作を確かめてから、完成したプログラムを BASTOC を使い C 言語に変換する。

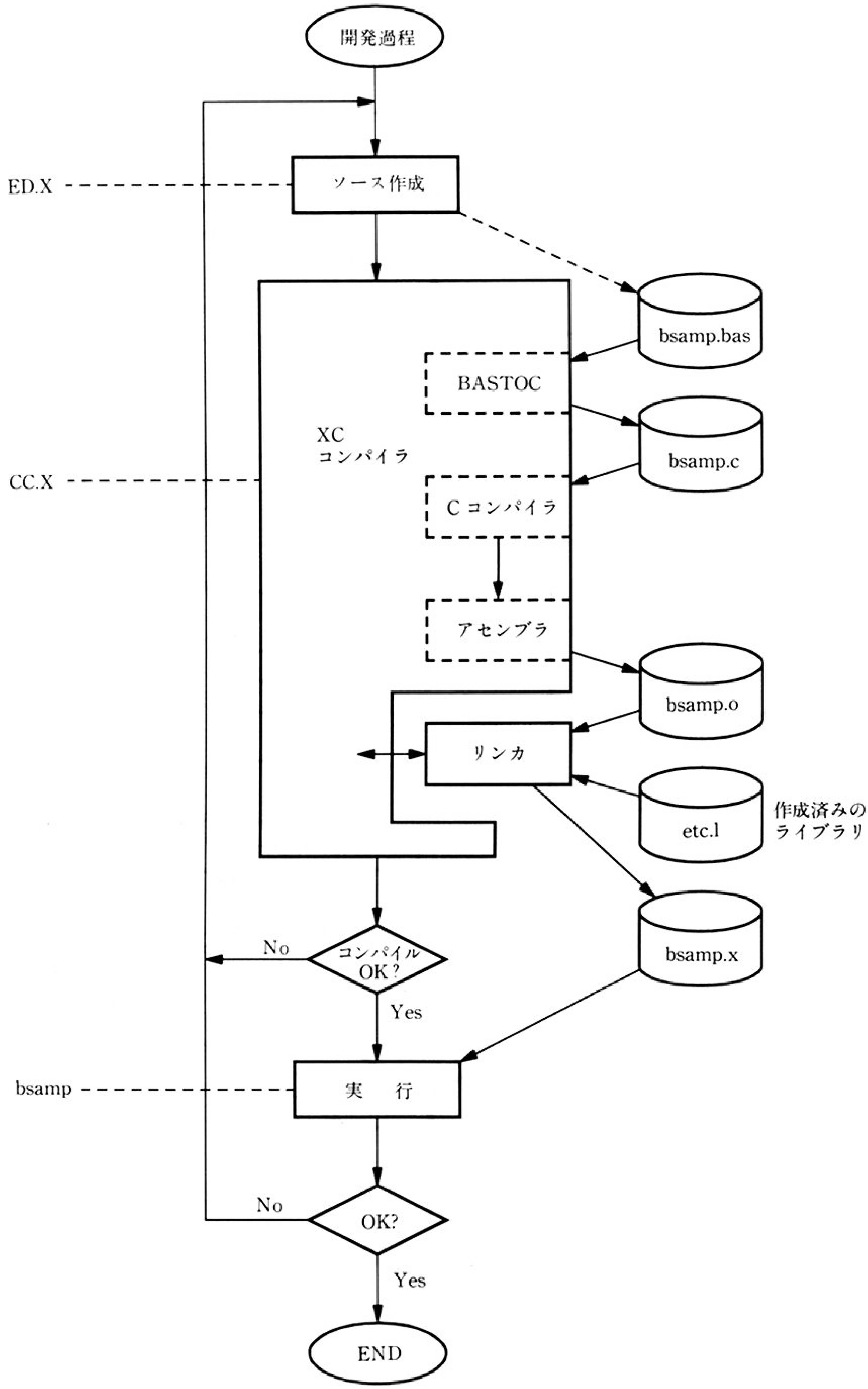
(2) X-BASIC のプログラムファイルを直接、CC コマンドに指定して、実行可能ファイルを作成してからデバッグ作業を行う。

このような2通りの方法がありますが、通常は、(1)の方法を採用してください。

5.2 プログラムの作成の流れ



5.2 プログラムの作成の流れ



5.3 変換方法

BASTOCのコマンド名は、"BC"で¥BCのディレクトリ下にあります。
BCコマンドはCCコマンド同様、コマンドラインに入力します。

```
B>bc　bprog.bas ↵
```

上記の例は"bprog.bas"というX-BASICのプログラムをBCコマンドラインに指定した例です。
この場合、出力は"bprog.c"というC言語のプログラムになります。
また、ファイル名に拡張子をつけてもかまいませんが、その場合でも出力ファイルは. cという拡張子になります。

5.3.1 コマンドラインの書式

BCのコマンドラインの書式を説明します。

```
bc[<スイッチ>]ソースファイル名[<オブジェクトファイル名>]
```

ソースファイル名
ソースファイル名の指定は、OSに依存します。
「2.1.2　コマンドラインの指定」を参照してください。
なお、CCコマンドで変換する場合は、拡張子を. basあるいは. bにしなくてはなりません。

オブジェクトファイル名
オブジェクトファイル名はオプションです。
オブジェクトファイル名の指定もやはり、OSに依存します。
また、オブジェクトファイル名はCON、PRN、AUXを指定してもかまい

ません。
拡張子はつけてもつけなくてもかまいませんが、このBCのコマンドの次のフェースがCCコマンドであることから、.cをつけるのが一般的です。

スイッチ

スイッチの指定もオプションです。
スイッチは、BCコマンドを実行するうえで、ユーザーに変換時の模様や出力ファイルへのコメント行(BASIC各行)の挿入を行います。
次に各スイッチを説明します。

／C＜行頭TAB数＞

機　能

オブジェクトファイルにBASICの各行をC言語の注釈として出力します。
＜行頭TAB数＞を指定することにより、注釈の前のTABの数を指定することができます。
＜行頭TAB数＞を省略した場合は、7個のTABが挿入されます。

省　略

注釈を出力しません。

例

```
B>bc　／C10　bprog.bas ⏎
```

/D

機　能　BCの変換の様子を色つきで標準出力に表示します。
リダイレクションを指定した場合は、色が表示されません。
各色は、次のような意味をもちます。

色	意　味
水色	行番号、数値
黄色	式、記号
白	未定義のもの
水色（強調）	変数、FUNC
黄色（強調）	DFFファイル（.def）で定義している文や関数
白（強調）	BCが内部処理する文や関数
水色（反転）	注釈、文字列
黄色（反転）	エラー（書式の誤り等）
白（反転）	#C、#ENDCで囲んであった部分

省　略　本機能は実行されません。

例

```
B>bc  /D  bprog.bas ⏎
```

／V

機　　能　現在コンバート中のX-BASICのソースプログラムを標準出力に表示します。これは、現在どこをコンバートしているかを逐次表示する機能です。

省　　略　本機能は実行されません。

例

```
B>bc　／V　bprog.bas ⏎
```

/E

機　　能　通常は、エラーをコメント(／＊～＊／)の型でファイルに出力しますが、このスイッチでエラー行を標準出力に表示するようにします。

省　　略　標準出力にエラー行を表示しません。

例

```
B>bc  /E  bprog.bas ⏎
```

／H＜ハッシュバッファの数＞

機　能

BCは、ハッシュテーブルを使用して関数・変数を管理していますが、大きなプログラムでは、このテーブルをオーバーすることがあります。
BCが次のエラーを出力して中断したときは、このスイッチを使用してハッシュバッファの数を増やして、再度やりなおしてください。
ハッシュバッファの数は、1～32を指定します。
実際の個数は、指定した数の1997倍の個数になります。

省　略

1を指定したことになり、ハッシュバッファは1997個で処理しています。

ハッシュバッファ不足のメッセージ

```
ハッシュバッファが足りません　／Hスイッチで確保してください
```

例

```
B>bc　／H2　bprog.bas ↵
```

5.3.2 CC コマンドでの変換

前にも述べたように、BASTOC は CC コマンドでも使用できます。
CC コマンドのファイル指定のところを、BASIC ソースプログラムのファイル名を指定します。
なお、この BASIC のファイル名の指定に関しては、必ず拡張子. bas あるいは. b をつけなければなりません。
. bas あるいは. b をつけることにより、C コンパイラは、指定された入力ファイルが X-BASIC のソースプログラムであることを認識して、X-BASIC から C への変換を行います。
またリンク時に X-BASIC 固有の関数を BASLIB. L に探しにいきます。
X-BASIC 固有の関数は、次の節で紹介します。
この C コンパイラで変換する方法を使用することにより、ユーザーは、いちいち BC コマンドを実行してから、CC コマンドを実行するというような、2度の手間を1度で済ませることができるわけです。

```
B>cc  bprog.bas ↵
```

CC コマンドの変換を行う場合には、BC コマンドのスイッチを指定することはできません。

5.3.3 BASTOC のための C 言語用ライブラリ関数

BASTOC のための C 言語用ライブラリ関数には次のような関数が用意されています。
詳しい内容については「C ライブラリマニュアル VOL. 1」を参照してください。

5.3 変換方法

abs	a_cont	a_end	apage	a_play	a_rec
asc	a_stat	a_stop	atan	atof	atoi
b_binS	b_chrS	b_csw	b_dateS	b_dayS	beep
b_exit	b_fclose	b_fcloseall	b_feof	b_fgetc	b_flprint
b_fopen	b_fprint	b_fputc	b_fread	b_freads	b_free
b_fseek	b_fwrite	b_fwrites	bg_fill	bg_get	
bg_put	bg_scroll	bg_set	bg_stat	b_hexS	b_ilprint
b_init	b_inkey0	b_inkeyS	b_input	b_int	b_iprint
b_itoa	b_leftS	b_linput	b_midS	b_mirrorS	b_octS
box	b_pi	b_rightS	b_setdate	b_settime	b_slprint
b_spaceS	b_spaceS	b_sprint	b_stradd	b_strchr	b_strcmp
b_strfS	b_striS	b_stringS	b_strncpy	b_strrchr	b_strtok
b_timeS	b_tlprint	b_tpalet	b_tprint	circle	child
cls	color	console	contrast	cos	crt
csrlin	dskf	ecvt	exp	fabs	fcvt
fdelete	fill	fix	frename	gcvt	get
home	hsv	img_color	img_home	img_ht	img_load
img_pos	img_put	img_save	img_scrn	img_set	img_still
instr	isalnum	isalpha	isascii	iscntrl	isdigit
isgraph	islower	isprint	ispunct	isspace	isupper
isxdigit	key	keysns	line	locate	log
m_alloc	m_assign	m_cont	md_cont	md_init	md_off
md_on	md_play	md_regr	md_regw	md_stat	md_stop
md_wrt	m_free	m_init	mouse	m_play	msarea
msbtn	mspos	msstat	m_stat	m_stop	m_tempo
m_trk	m_vget	m_vset	paint	palet	pi
point	pos	pow	pset	put	rand
randomize	rgb	rnd	screen	setmspos	sgn
sin	sp_clr	sp_color	sp_def	sp_disp	sp_init
sp_move	sp_off	sp_on	sp_pat	sp_set	sp_stat
sqrt	srand	stick	strcspn	strig	strlen
strlwr	strnset	strrev	strset	strspn	strupr
symbol	tan	toascii	tolower	toupper	using
val	v_cut	vpage	width	window	wipe

5.4 BASTOCでのBASICプログラムの書きかた

本節ではBASTOCを使用するうえで、変換しやすいプログラムの説明をします。

5.4.1 名前のつけかた

BASTOCを利用する場合、関数名やサブルーチン名などのつけかたに制限があります。
BASTOCで変換する際に、ある規則に従って変換するため、使ってはいけない名前や、注意しなければならない名前があります。
これを無視した場合、コンパイル時にエラーが発生する可能性があります。

(1) 使ってはいけない名前

① main
関数名に"main"を使用してはいけません。

② S ######
######は6桁までの半角数字で、この名前はgosub文に対応するラベル名として使用されます。

③ L ######
######は6桁までの半角数字で、この名前はgoto文に対応するラベル名として使用されます。

④ b_??????
"b_"で始まる名前は使ってはいけません。
これは、BASTOCが??????の部分をある規則に従って、BASICで使っている名前をそのまま使用するためです。

(2) 注意の必要な名前

$で終わっている名前が、組み込み関数名やシステム関数名ではない場合、BASTOCは次の規則に従って変換します。

(3) 規　則

名前の最後の'$'を大文字の'S'に置き換える。

```
  ↓
bun$()
  ↓
```

BASIC
プログラム

```
  ↓
bunS();
  ↓
```

C言語
プログラム

上記のような規則に従って変換されるため、BASICプログラムではbun$の有効範囲でbunSを使えません。

5.4.2 式についての注意

演算順序をはっきりさせること

X-BASICの式をそのまま変換した場合、一般的な算術式の範囲内では問題ありませんが、シフト演算、ビット演算、比較演算、論理演算が混在している場合は、演算の順序を明確にするために、カッコを使って記述してください。

```
if  a+b<>0  then  x=a+b
```

```
if  (a+b)<>0  then  x=a+b
```

5.4.3 論理演算式についての注意

(1)　not の使い方

C 言語と X-BASIC では、not の優先順位が異なります。
not を使う場合は、カッコを使用したり、not の有効範囲を明確に示す必要があります。
これを無視した場合、結果が違ってきます。

```
if not(a=0) or b=0 then x=0
```

(2)　ビット論理演算

ビットごとの論理演算では、and、or、xor が使えます。

(3)　論理演算の値について

X-BASIC での論理演算の値は"−1"か"0"ですが、C 言語では"1"か"0"です。
この値を積極的に使った場合(数値として用いた場合)、X-BASIC と XC ではプログラムの動きが変わることがあります。
論理値の判定には 0（偽）か、0 でない（真）かで行ってください。

5.4.4 してはいけないこと

① endfunc の後に main 部のプログラムを続けて書くことはいけません。
② func で宣言した関数の中から、関数の外に goto 文で飛び出してはいけません。
③ func で宣言した関数を、gosub 文で呼び出してはいけません。
④ 次の関数の引数に、文字列定数を指定してはいけません。
　strlwr
　strupr
　strrev
⑤ プログラムの最後の endfunc 文の後に、注釈文、または空文以外の文を書いてはいけません。

⑥ switch 文で case の後に文字列を書いてはいけません。

⑦ 論理値を数値として使ってはいけません (0か、非0として使うのはよい)。

例

```
char c:  int k, l: float a
k=isdigit(c) : if k=-1  then print  "Yes" else print  "No"
  l=-(a>3.2)* 10-3 *(a>5.6)
```

上記のようなプログラムは、次のようなプログラムにしてください。

```
char c : int k,l : float a
k=isdigit(c) : if k<>0 then print  "Yes" else print  "No"
  l=0 : if a>3.2 then  l=l+10 : if a>5.6 then  l=l+3
```

⑧ key または list 文を使ってはいけません。

5.4.5 新たにできること

① b_argc,b_argv()を使って、コマンドラインをとり込むことができます。

例

```
int c,i
c=b_argc
for i=0  to  c-1
  print b_argv(i)
next
```

5.5 BASTOCのファイル構成

本節では、BASTOCを使用するためのファイル構成について説明します。
XCコンパイラのパッケージには、このファイル構成についての知識がなくても、すぐBASTOCコマンドが使用できるように、セッティングされていますので、初心者のかたはこの節を読み飛ばしてもさしつかえありません。

5.5.1 ディレクトリ

ルートディレクトリ
- BC
  このディレクトリの下にはBASTOCの本体である実行可能ファイル(BC. X)や、変換に必要なファイルがあります。
- INCLUDE
  CCとBASTOC共用のインクルードファイルが納められています。

5.5.2 実行可能ファイル

BASTOCの実行可能ファイルは、BCディレクトリの下にあるBC.Xだけです。

5.5.3 CNFファイル

CNFファイルとは、OS起動時のCONFIG. SYSのようなファイルです。
X-BASICは拡張型のBASICで、BASIC本体にさまざまな機能を付加することができます。
通常、この機能はX-BASICの起動時に決定されており、この関数の付加情報を記述してあるのが、BASIC. CNFです。
BASIC. CNFはテキストファイルですので、エディタで修正することができます。

5.5.4 DEF ファイル

X-BASICがもともと用意してある関数、mid$()、sin()や、XXXXX.FNCとして後から組み込む関数、a_rec()などは、XCの組み込み関数ではありません。
そこで、C言語用に書き換える必要があります。
そのための指示が、このDEFファイルに記述されています。
DEFファイルは、各ライブラリごとに.DEFという拡張子をもったファイルとして、BCディレクトリの下に登録されています。
次に各DEFファイルの一覧を示します。

BASIC.DEF
MUSIC2.DEF
SPRITE.DEF
AUDIO.DEF
MOUSE.DEF
STICK.DEF
GRAPH.DEF
IMAGE.DEF

(1) DEF ファイル形式

DEFファイルの形式は次のように、":"で区切られています。
左側にX-BASICの関数の形式、右側に左側の関数に対応するXCの関数を記述します。

X-BASICの関数形式：XCの関数形式

例

```
1    fread(NA, I, I)      : b_fread(%,@,%,%)
F    atof(S)              : (%)
S    mid$(S,I,I)          : b_midS($,%,%,%)
1    msstat(I,I,I,I)      : (&,&,&,&)
```

(2) BASIC 関数の形式

BASIC 関数の記述形式を説明します。

<関数の型><関数名>(<引数の型指定並び>)

●関数の型……型記号を用いて型指定します。

●型指定………下記に示す型記号、または、それに“—”をつけたもの。
<型記号>|<型記号>—

●型記号型

C	文字型
I	整数型
F	実数型
S	文字列型
CA	文字型配列
IA	整数型配列
FA	実数型配列
NA	数型配列(文字型、整数型、実数型のいずれでもよい)
SA	文字列型配列

●関数名……BASIC 関数名

●引数の型指定並び……引数の型が上記の型記号で示されています。

例

```
F    atof(S)
I    fread(NA,I,I)
S    mid$(S,I,I)
I    msstat(I,I,I,I)
```

(3) C 関数の形式

C 関数の記述形式を説明します。

```
[<関数名>](<引数の指定並び>)
```

● 関数名……対応するCの関数名
　　省略するとX-BASICの関数名と同じ名前が使われます。

● 引数の指定並び
　　<引数の指定> | <引数の指定並び><引数の指定>

● 引数の指定
　引数の指定とは、たとえば、a_rec(NA,I)というBASIC関数が、XCではa_rec(buff,sizeof(buff),cl)となるように、実引数を変換するときの方法を指定するものです。
　引数の指定には、次のような記号があります。

………領域の大きさが必要な場合、指定します。
#は“sizeof(1つ前の引数)”として変換されます。
1つ前の実引数は単純変数か、配列でなければなりません。
XCの関数側では第1引数以外で指定してください。

@………領域の要素の大きさが必要な場合、指定します。
@は“sizeof(1つ前の引数[0])”として変換されます。
1つ前の実引数は配列か文字変数でなければなりません。
XCの関数では第1引数以外で指定してください。

&………実引数の変数の内容を変える場合、指定します。
実引数の変数の内容を変える場合、XCでは引数のアドレスを関数にわたさなければならず、引数に&演算子を付加します。
また、X-BASICの実引数は単純変数になっていなければなりません。

$………$はmid$(“abC”,2,1)などのX-BASIC特有の文字列関数をXCで実現させるときのみ用いるための指示です。
strtmp0,　strtmp1,……………strtmp9を実引数として指定する場合、必ずXCの関数の引数として指定しなければなりません。

%………%はX-BASICの形式をそのまま変換します。
例

```
(%)
b_fread(%,#,%,%)
midS($,%,%,%)
(&,&,&,&)
```

※単純変数とは
単純変数とは、配列変数ではない変数のことをいいます。

① DIMで宣言していない変数(単純変数)
② STRで宣言している変数が文字列変数(単純変数)
③ DIM STRで宣言している変数が文字列配列(配列)

5.5.5 インクルードファイル

インクルードファイルはXCコンパイラと共用するため、「1.2.2 (2) INCLUDE」の中に含まれています。
BASTOCのインクルードファイルは次の通りです。

AUDIO.H
BASIC.H
BASIC0.H
GRAPH.H
IMAGE.H
MOUSE.H
MUSIC.H
MUSIC2.H
SPRITE.H
STICK.H

5.5.6 外部関数

ユーザーが独自に作成した外部関数を使用したプログラムは、単にBASTOCを使用するだけでは実行可能ファイルにすることはできません。
ここでは、ユーザーが作成した外部関数を実行可能ファイルにするための手順について説明します。

①X－BASICで完全に動作するBASICのソースプログラムを作成する。
②外部関数に対応した、DEFファイルを作成する。
DEFファイルの構造については、「5.5.4　DEFファイル」を参照してください。
新しくファイルを作成しても、すでにあるDEFファイルに付け加えてもかまいません。
ただし、DEFファイルはBC.Xと同じディレクトリにある必要があります。
③外部関数に対応するCのライブラリを作成する。
具体的には、外部関数のソースプログラム（アセンブリ）から関数単位で実行部分を抜き出し、C言語とのインターフェース部分を付け加えます。
詳しくは、「第4章　アセンブラとのインターフェース」を参照してください。
また、新しくC言語を使用して同機能の関数を作成してもかまいません。
④③で作成したプログラムをアセンブルする。
このときリンクする必要はなく、オブジェクトファイル（.O）のままでかまいません。
⑤BASLIB.Lに登録する。
④で作成したオブジェクトファイル（.O）をLIB.Xを使用して、BASLIB.Lに登録します。
使用頻度の低い関数はライブラリとして登録する必要はありません。
ただし、そのときはリンクするためファイル名を指定しなければなりません。
また、既存のライブラリに登録するときは、同じ名前のファイルがないかを確認してください。
ライブラリの登録ファイル名はLIB.Xの／Lスイッチで知ることができます。

⑥インクルードファイルで関数宣言をする。

新しく作成した外部関数のライブラリを利用するため、インクルードファイル（BASIC. HかBASIC0. H）内で関数宣言を行います。

関数宣言については「Cリファレンスマニュアル」の「3. 2　関数」を参照してください。

関数宣言は、Cのソースプログラム内でしてもかまいません。

また、別のインクルードファイルで宣言をしてもかまいません。

しかし、それにはCのソースプログラムを書き換えなければならいので、CC. Xを使用して一度に実行可能ファイルを作成することはできなくなります。

⑦コンパイルする。

CC. Xを使用してコンパイルします（. BAS　→　. X）。

あらかじめ、BC. Xを使用してCのソースプログラムを作成してからコンパイルしてもかまいませんが、コンパイル時にBASLIB. Lをリンクさせるため CC. Xの／Wスイッチを指定してください。

⑤でライブラリファイルに登録していない場合は次のように、コマンドラインでリンクするファイル名を指定してください。

```
A>CC  SAMPLE. BAS  SAMPLE1. O
```

⑧完成

まとめると、DEFファイル、インクルードファイル、ライブラリの3つを用意し、CC. Xを使用するだけで、外部関数を用いたBASICプログラムを実行可能ファイルにすることができます。

そのうち、DEFファイルとインクルードファイルは、関数1個あたり1行の宣言を行うだけです。

ライブラリもX－BASICとのやりとりをするインフォメーションテーブルや、トークンテーブルなどのインターフェース部を、C言語とのインターフェース部に置き替えるだけでよいので、比較的簡単に移植することができます。

第6章

追加 BASIC 関数

追加関数リファレンス

MIDI 拡張 MML

サンプルプログラム

X68000ではMIDIボードを実装することで、MIDIインターフェイスが装備されている市販の楽器を制御でき、内蔵音源を越えたすばらしい演奏が可能になります。
XCコンパイラは、“OPMDRV2.X”（デバイスドライバ）や“MUSIC2.FNC”（インクルードファイル）を組み込むことで、MIDIボードの制御が可能になります。
本章では、これにともなって追加されたBASICの命令リファレンスと、拡張されたMMLについて説明します。
これにより、BASICでMIDIを使った演奏が可能になり、BASICプログラムコンバータを用いることで、C言語でも演奏が可能になります。
また、サンプルプログラムも章末に掲載してあります。

6.1 追加関数リファレンス

MD_CONT　MUSIC

書式	MD_CONT (n)
引数	int
戻り値	int
機能	指定したチャンネルのFM音源、およびMIDIの演奏を再開します。 この命令は、MD_STOPで一時中断した演奏を再開するときに使います。 nを省略すると、すべてのチャンネルの演奏を再開します。 0~7ビット　：　FM音源の1~8チャンネル 8~23ビット　：　MIDIの9~24チャンネル ビットを1にしたチャンネルを再開する 戻り値として、正常のときは0、エラーのときは−1を返します。

MD_INIT　MUSIC

書式	MD_INIT ()
引数	MIDIボードの初期化をします。 M_ASSIGNで割りあてたトラックは、MD_INITで初期化してください。 また、M_ASSIGNで割りあてる9~24チャンネルは、それぞれデフォルトで81~96トラックに対応しています。

MD_OFF

MUSIC

書　　式	MD_OFF (n)
引　　数	int
戻 り 値	int
機　　能	指定したMIDIのチャンネルの出力を停止します。 nを省略すると、全チャンネルの出力を停止します。 戻り値として、正常のときは0、エラーのときは−1を返します。

0〜7ビット	：	常に0
8〜23ビット	：	MIDIの9〜24チャンネル ビット1にしたチャンネルの出力を停止する
24〜31ビット	：	常に0

MD_ON

MUSIC

書　　式	MD_ON (n)
引　　数	int
戻 り 値	int
機　　能	指定したMIDIのチャンネルの出力を開始します。 nを省略すると、全チャンネルの出力を開始します。 戻り値として、正常のときは0、エラーのときは−1を返します。

0〜7ビット	：	常に0
8〜23ビット	：	MIDIの9〜24チャンネル ビット1にしたチャンネルの出力を開始する
24〜31ビット	：	常に0

MD_PLAY

MUSIC

書　式	MD_PLAY (n)
引　数	int
戻り値	int
機　能	指定したチャンネルの FM 音源、および MIDI の演奏を開始します。 n を省略すると、すべてのチャンネルの演奏を開始します。 0～7 ビット　：FM 音源の 1～8 チャンネル 8～23 ビット　：MIDI の 9～24 チャンネル 　　　　　　　　ビットを 1 にしたチャンネルを演奏する 戻り値として、正常のときは 0、エラーのときは－1 を返します。

MD_REGR

MUSIC

書　式	MD_REGR (n, m)
引　数	char
戻り値	int
機　能	MIDI 制御 IC (YM3802) のレジスタを読み込みます。 n にグループナンバー、m にはレジスタナンバーを指定します。 戻り値として、正常のときは 0、エラーのときは－1 を返します。 注：間違ったパラメータを指定すると、システムが正常に動作しなくなる場合があります。 YM3802 の機能を十分に理解の上でご使用ください。

MD_REGW

書式	MD_REGW (n, m, dt)
引数	char
戻り値	int
機能	MIDI 制御 IC (YM3802) のレジスタに値を書き込みます。 n にグループナンバー、m にはレジスタナンバーを指定します。 dt にはレジスタに書き込む値を、0～255 の範囲で指定します。 戻り値として、正常のときは 0、エラーのときは－1 を返します。 注：間違ったパラメータを指定すると、システムが正常に動作しなくなる場合があります。 YM3802 の機能を十分に理解の上でご使用ください。

MD_STAT

MUSIC

書式	MD_STAT (ch)
引数	char
戻り値	int
機能	chで指定したMIDIのチャンネルへ、データが出力できるかどうかを調べます。 chには、状態を調べたいチャンネルを9～24で指定します。 chを省略すると、すべてのチャンネルの状態を調べます。

・chが9～24のとき

0 ： 出力停止
1 ： 出力可能

・chを省略したとき

0～7ビット ： 常に0
8～23ビット ： MIDIの9～24チャンネル
ビットが0で出力停止、1で出力可能
24～31ビット ： 常に0

どちらの場合もエラーのときは、－1を戻り値として返します。
MD_STATは演奏中かどうかを調べるものではありません。これについてはM STATを参照してください。

MD_STOP

MUSIC

書式	MD_STOP (n)
引数	int
戻り値	int
機能	指定したチャンネルのFM音源、およびMIDIの演奏を一時中断します。 nを省略すると、すべてのチャンネルの演奏を一時中断をします。

0〜 7ビット ： FM音源の1〜8チャンネル
8〜23ビット ： MIDIの9〜24チャンネル
ビットを1にしたチャンネルを一時中断する

戻り値として、正常のときは0、エラーのときは−1を返します。

MD_WRT

MUSIC

書式	MD_WRT (n)
引数	char
戻り値	int
機能	MIDIボードにnを出力します。nの範囲は0〜255です。 戻り値として、正常のときは0、エラーのときは−1を返します。

M_STAT

MUSIC

書式	M_STAT (ch)
引数	char
戻り値	int
機能	指定されたチャンネルが、演奏中かどうかを調べます。 ch には、調べたいチャンネルを1～24で指定します。 ch を省略すると、すべてのチャンネルの演奏状態を返します。

・ch が1～24のとき

0 ： 停止中
1 ： 演奏中

・ch を省略したとき

0～7 ビット	：	FM音源の1～8チャンネル
8～23 ビット	：	MIDIの9～24チャンネル ビットが0で停止中、1で演奏中
24～31 ビット	：	常に0

どちらの場合もエラーのときは、－1を戻り値として返します。
なお、この命令は従来のM_STATの機能を拡張したものです。

6.2 MIDI拡張MML

MIDI制御のために拡張になったMMLは、次の通りです。

MIDI拡張MML一覧

MMLデータ	意　　味	パラメータ範囲
'(アポストロフィー)	MIDI拡張MMLの使用を開始／終了する	
Tn	MIDI送信チャンネルセット	1～16
Pn	MIDIプログラムチェンジ	1～128
On, m	ノートオン	0～127
Fn	ノートオフ	0～127
$n	ダイレクト送信データ	0～255
n	ダイレクト送信データ	0～255

MIDIのベロシィティ（音量）は、MMLのVでだいたいの調節を、@Vで微調整を行います。
なお、デフォルトではV=8、@V=80に設定されています。
MIDI拡張MMLを使用する場合には、MIDI拡張MMLの先頭と末尾を　'　で囲まなくてはなりません。

例　o4'T1'c4d8e8f8g8a4

また、MIDI拡張MMLを連続して記述する場合は、各MMLデータを','で区切ってください。

例　o4'T1, P123, F40'c4d8e8a4

これら以外に次の3点も拡張されています。
・演奏中にCTRL+Dを押すと、演奏を停止します。
・トラックナンバーの最大値は223に増設されています（従来は80）。
・MIDIクロックは常に出力されていますので、外部のMIDI音源との同期が可能です。

6.3 サンプルプログラム

プログラム例

```
 10 /*
 20 /* music sample
 30 /* "おお、スザンナ！！"   S.C.Foster
 40 /*
 50 str tr,as,bs,cs,ds,part[255]
 60 m_init()
 70 md_init()
 80 m_alloc(10,2000)
 90 m_assign(10,10)
100 linput "ＭＩＤＩ送信チャンネル（１～１６）－－－＞？";tr
110 linput "プログラムナンバーを入力して下さい（１～１２８）－－－＞？";as
120 linput "テンポを入力して下さい（３２～２００）－－－＞？";bs
130 print
140 part="'T"+tr+",P"+as+"'"+" o4 v9 q7 l16"+" t"+bs
150 wr_trk(part)
160 part="g&a b8<d8d8e8> <d8>b8g8.a b8b8a8g8 a4.ga&"
170 wr_trk(part)
180 part="b8<d8d8.e> <d8>b8g8.a& b8b8a8a8 g4.ga&"
190 wr_trk(part)
200 part="b8<d8d8.e> <d8>b8g8.a b8b8a8g8 a4r8ga&"
210 wr_trk(part)
220 part="b8<d8d8e8> <d8.>bg8.a bb8.a8.a g4r4"
230 wr_trk(part)
240 part="<c4c4> <e8e4e8> <d8.d>b8g8 a4r8ga&"
250 wr_trk(part)
260 part="b8<d8d8.e> <d8>b8g8a8 b8b8a8.a g4r8"
270 wr_trk(part)
280 md_play()
290 linput "演奏を停止しますか＜ｙ／ｎ＞－－－＞？";cs
300 if cs="n" then end
310 md_stop()
320 linput "演奏を再開しますか＜ｙ／ｎ＞－－－＞？";ds
330 if ds="y" then md_cont()
340 end
350 func wr_trk(part;str)
360 m_trk(10,part)
370 endfunc
```

付　録

コントロールコード表
キャラクタコード表
コンパイルスイッチ一覧
エラーメッセージ一覧

1 コントロールコード表

次に示すコントロールコードをコンソール（画面）に対して出力することにより、画面を制御することができます。
コントロールコードは1文字です。

記号	コード(16進)	機能
BEL	07	Sound bell ベルを鳴らす ベルの音は、CONFIG.SYSファイルの中の“BELL=<ファイル名>”の記述に従う
BS	08	Cursor backward カーソルを1文字左に移動する カーソルが行の1カラム目にあるときは1行上の最終カラムに移動する カーソルがホームポジション(1カラム目、1行目)にあるときは無効
HT	09	Skip to next tab stop カーソルを次のタブ位置に移動する タブの位置は、キャラクタ座標の最大カラム数を超えない8の倍数で、次のように決められる。 08、16……キャラクタ座標の最大カラム数より小さい8の倍数の最大値(最大のタブ位置) カーソルが最大のタブ位置のカラムよりも右側にあるときは1行下の1カラム目に移動する カーソルが最終行にあるときは1行スクロールアップする
LF	0A	Cursor down 同じカラム位置でカーソルを1行下に移動する カーソルが最終行にあるときは1行スクロールアップする
VT	0B	Cursor up 同じカラム位置でカーソルを1行上に移動する カーソルが画面の1行目にあるときは無効

1. コントロールコード表

記号	コード(16進)	機　能
FF	0C	Cursor forward カーソルを1文字右に移動する カーソルが行の最終カラムにあるときは1行下の1カラム目に移動する カーソルが画面最終行の最終カラムにあるときは1行スクロールアップする
CR	0D	Cursor to left margin カーソルを行の1カラム目に移動する
SUB	1A	Clear screen 画面の表示をすべてクリアする その後、カーソルはホームポジションに移動する
ESC	1B	Introduce on ESC sequence エスケープコードの始まり ESCコードに続く文字により、エスケープシーケンスによる画面制御を行う
RS	1E	Cursor HOME カーソルをホームポジションに移動する

2 キャラクタコード表

次の表は、1バイトコード(半角文字)の一覧です。

1バイトコード文字には、キャラクタコードの0～255(16進法で0x00～0xFF)が割りあてられています。

キャラクタコードに割りあてられている文字には、図形文字と制御文字があります。

図形文字は、画面上の表示やプリンタの印刷のための文字であり、制御文字は画面の制御とプリンタの制御のための文字です。

プリンタに対する制御文字の働きについては、各プリンタの取扱説明書を参照してください。

また、0x80～0x9F、0xE0～0xFFまでは、漢字などの2バイトコード文字の1バイト目として使用されます。

上位4ビット→

下位4ビット↓

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0				0	@	P	`	p				ー	タ	ミ		
1			!	1	A	Q	a	q			。	ア	チ	ム		
2			"	2	B	R	b	r			「	イ	ツ	メ		
3			#	3	C	S	c	s			」	ウ	テ	モ		
4			$	4	D	T	d	t	2バイトコード文字の1バイト目		、	エ	ト	ヤ	2バイトコード文字の1バイト目	
5			%	5	E	U	e	u			・	オ	ナ	ユ		
6			&	6	F	V	f	v			ヲ	カ	ニ	ヨ		
7	制御文字		'	7	G	W	g	w			ァ	キ	ヌ	ラ		
8			(	8	H	X	h	x			ィ	ク	ネ	リ		
9			)	9	I	Y	i	y			ゥ	ケ	ノ	ル		
A			*	:	J	Z	j	z			ェ	コ	ハ	レ		
B			+	;	K	[	k	{			ォ	サ	ヒ	ロ		
C			,	<	L	¥	l	|			ャ	シ	フ	ワ		
D			−	=	M	]	m	}			ュ	ス	ヘ	ン		
E			.	>	N	^	n	‾			ョ	セ	ホ	ﾞ		
F			/	?	O	_	o				ッ	ソ	マ	ﾟ		

3　コンパイルスイッチ一覧

スイッチ名	機　　能	ページ
／A	警告レベルの指定	64
／B	C 言語ソースファイル（.c）のみの作成	65
／C	コメント行の保存	47
／D	マクロ定義	48
／E	識別子の最大長の指定	66
／Fc	リンクの抑制	53
／Fd	アセンブラソースファイル（.s）の出力ファイル名の指定	54
／Fo	オブジェクトファイル（.o）の出力ファイル名の指定	55
／Fs	アセンブラソースファイル（.s）のみの作成	56
／Fx	実行可能ファイル（.x）の出力ファイル名の指定	57
／G0	シンボルの最大個数の指定（パーザ部）	72
／Ga	シンボルの最大個数の指定（AS 部）	73
／Gb	ハッシュバッファの最大個数の指定（BC 部）	74
／Gc	スタック検査の指定	61
／Gh	ヒープサイズの設定	62
／Gp	シンボルの最大個数の指定（プリプロセッサ部）	75
／Gs	スタックサイズの設定	63
／I	インクルードパスの指定	49
／J	char 型を unsigned char とする	67
／Na	32K バイト以上の auto 変数の使用許可	76
／Nf	浮動小数点演算ライブラリの変更	58
／Nl	ライブラリパスの指定	59
／Ns	ソースコードデバッガ（SCD. X）用コードの生成の指定	60
／O	プログラムの最適化	46
／P	プリプロセッサの出力をファイルへ出力	50
／Q	同一コード文字列の圧縮	68
／T	作業パスの指定	69
／U	マクロ定義の取り消し	51
／V	プリプロセッサの出力を標準出力へ出力	52
／W	BASIC ライブラリの使用フラグ	71
／X	定義済み識別子__STDC__を未定義	77
／Y	DOS／IOCS ライブラリの使用フラブ	70

4 エラーメッセージ一覧

エラーメッセージ	意　　味
CC ドライバのエラー	
CTRL^C Terminated	CTRL+C またはアボートで中断されました。
Can't find（実行ファイル）	起動しようとした実行ファイルが見つかりません。
CC driver error : No enough memory	メモリが足りません。
Fatal Error in（エラー番号）	エラーが発生しました。エラー番号は errno が示すエラー番号です。
CC ドライバのワーニング	
illegal option string(文字列)is ignore	不正なオプションスイッチです。無視しました。
illegal file name(ファイル名)is ignore	不正なファイル名です。無視しました。
CC driver warning : option duplicate	同じオプションスイッチを2回以上指定しています。
CC driver warning : illegal option	不正なオプションスイッチが指定されています。
CC driver warning : illegal option used	オプションスイッチの使用方法が不正です。
XBAStoC のエラー	
source file not found	指定の入力ファイルが見つかりません。
source file open error	指定の入力ファイルがオープンできません。
source file read error	指定の入力ファイルから入力できません。
destination file write error	指定の出力ファイルに出力できません。
no enough memory	メモリが足りません。
symbol table overflow	シンボル情報があふれました。
#endc less	#endc がありません。
illegal statement number	行番号に誤りがあります。
syntax error	構文に誤りがあります。
FOR-NEXT initialize error	for 文の初期値設定に誤りがあります。

4.エラーメッセージ一覧

エラーメッセージ	意　　味
illegal GOTO/GOSUB	goto 文，gosub 文に誤りがあります。
too few parameters in call	引数の数が違います。
illegal data type	データの型が違います。
statement without main program	end 文以降に無効な文があります。
statement without function	endfunc 文以降に無効な文があります。
nothing THEN of IF statement	if 文の then がありません。
nothing format string of USING statement	using 文の書式制御文字列がありません。
nothing semicolon of USING statement	using 文の書式制御文字列の次に';'がありません。
illegal SWITCH-ENDSIWTCH statement	swich 文と endswitch 文の対応に誤りがあります。
illegal REPEAT-UNTILL statement	repeat 文と untill 文の対応に誤りがあります。
illegal WHILE-ENDWHILE statement	while 文と endwhile 文の対応に誤りがあります。
illegal FOR-NEXT statement	for 文と next 文の対応に誤りがあります。
illegal FUNC-ENDFUNC statement	func 文と endfunc 文の対応に誤りがあります。
{} error	{と} の対応に誤りがあります。
illegal data in {}	{...} の中のデータに誤りがあります。
nothing function name	func 文に関数の名前がありません。
() error	(と) の対応に誤りがあります。
undefined statement number	goto 文，gosub 文で指定した行番号がありません。
illegal statement number in GOTO/GOSUB	goto 文，gosub 文で指定した行番号が無効です。
function name duplicate	関数の名前は既に定義されています。
BC stack overflow	式が複雑すぎて評価できません。

4.エラーメッセージ一覧

エラーメッセージ	意　　味
プリプロセッサのエラー	
(マクロ名)：recursively macro	マクロ名は再帰呼び出しされています。
(マクロ名)：actuals too long	関数型マクロの定義で、実引数が長すぎます。
token too long	トークンが長すぎます。
マクロ名：unterminated macro call	マクロ呼び出しが完結していません。
too many defines	マクロ定義の数が多すぎます。
マクロ名：too much pushback	マクロのネストが最高値（14個）を越えました。
no space	メモリに空きがありません。
bad include syntax	# include 文の文法が違います。
Unreasonable include nesting	# include 文のネストが深すぎます。
Unreasonable if nesting	# if, # ifdef. # ifndef 文のネストが深すぎます。
cpp internal error : No Space	プリプロセッサの内部エラーです。メモリに空きがありません。
Can't find include file ファイル名：パス名	指定されたインクルードファイルが、パス名のディレクトリに見つかりません。
too much defining	マクロの定義内容が大きすぎます。
マクロ名：missing)	関数型マクロの定義で、）が足りません。
bad formal : ¥	関数型マクロの定義で、定義内容以外の部分に対して行の継続を行っています。
too much formals : 仮引数名	関数型マクロの定義で、仮引数の個数が多すぎます。
## operator undefined 1 paramater	##の左辺のパラメータが定義されていません。
# operator should be followed by a macro argument name	マクロの仮引数と # のパラメータが一致していません。
## operator undefined 2 paramater	##の右辺のパラメータが定義されていません。
too many/D options, ignoring 引数	/D オプションが20個を超えました。引数を無視しました。

4.エラーメッセージ一覧

エラーメッセージ	意　　味
too many/U options, ignoring 引数	/U オプションが20個を超えました。引数を無視しました。
excessive/I file（パス名）ignored	/I オプションが 8 個を超えました。パス名を無視しました。
No source file ファイル名	ソースファイルがありません。
Can't create ファイル名	作業用ファイルが作成できません。
# endif less	# endjf 文に対応する # if がありません。
# else less	# else 文に対応する # if がありません。
undefined control	プリプロセッサ命令以外の未定義な#で始まる文があります。
# line Illegal line no	# line 文に指定した行番号が不正です。
# line Illegal file name	# line 文に指定したファイル名が不正です。
Illegal character 文字 in preprocessor # if	# if 文、# elif 文の式中に不正な文字があります。
Illegal number 文字列	文字列は数字として評価できない不正な文字列です。
yacc stack overflow	式が複雑すぎて評価できません。
syntax error	式評価のエラー、またはプリプロセッサ命令の文法エラーです。
プリプロセッサのワーニング	
（マクロ名）：argument overflow	マクロの仮引数の数が多すぎます。
（マクロ名）：argument mismatch	マクロの仮引数の数があっていません。
Illegal macro name	マクロ名に不正な文字を使用しています。
マクロ名 redefined	同一のマクロ名で、異なる内容を定義しなおしています。
パーザの致命的エラー	
identifier overflow	識別子が多すぎます。
symbol table over	シンボル情報があふれました。
no enough memory	メモリが足りません。
expr stack overflow	式が複雑すぎて評価できません。

4.エラーメッセージ一覧

エラーメッセージ	意　　味
imdc file error	作業用ファイルの作成に失敗しました。
reach EOF	()、{}、""、/**/などが閉じていません。ファイルの終わりに達しました。
source file not found	ソースファイルが見つかりません。
temporary file create error	作業用ファイルが作成できません。
パーザのエラー	
Unknown character	不正な文字が存在します。
illegal preprocessor # line	プリプロセッサで処理されなかった#命令が残っています。
comment statement error	注釈文の誤りです。
string error	文字列が閉じていません。
undefined identififer in function name: 識別子	関数内で未定義の識別子が使用されています。
number syntax error	数値定数で許されない数字を使用しています。
expression error	式評価の誤りです。
storage class illegal：記憶クラス指定子	記憶クラスの指定が不正です。
too many }	} が多すぎます。
external definition error	外部参照宣言の誤りです。
illegal use of storage class：arg.	引数の宣言において記憶クラスの指定が不正です。
illegal use of storage class：proto type arg.	引数のプロトタイプ宣言において記憶クラスの指定が不正です。
illegal use of storage class：記憶クラス指定子	記憶クラス指定子の使用法が不正です。
illegal use of indirect：識別子	式中の間接演算子の使い方の誤りです。
identifier duplicate：識別子	識別子が重複しています。
intermediate cade error	中間コードの誤りです。
undefined struct/union name	構造体/共用体の名前が未定義です。
illegal type specifier	不正が型指定子が指定されています。
no assign array element	配列の添字数がありません。
too many initializer	初期化要素が多すぎます。

4.エラーメッセージ一覧

エラーメッセージ	意　　味
enum constant error	列挙型の定数の誤りです。
declaration error	宣言の誤りです。
missing]	］が足りません。
missing) in declarator	宣言において）が足りません。
missing) in function declaration	関数宣言において）が足りません。
function declaration error	関数宣言の誤りです。
{} error in struct/uion	構造体/共用体で｛と｝が合っていません。
overflow type specifier	型指定子が複雑すぎて、オーバーフローしました。
identifier redeclaration	識別子が再宣言されています。
struct/untion/enum tag name error：タグ名	構造体/共用体/列挙型のタグ名（名札）が誤りです。
storage class error at function	関数の記憶クラス指定が誤りです。
struct member overflow	構造体のメンバ要素があふれました。
break error	break 文の誤りです。
continue error	continue 文の誤りです。
cast type error	型変換において型指定子が誤りです。
difference typedef name	typedef で指定した名前が、式中に不正な使われかたをしています。
field width overflow	ビットフィールドのビット数があふれました。
fiele type error	1ビットフィールドのビット幅が、最高値を越えました。
array dimension overflow	配列の次元数があふれました。
element index mdssing	配列の要素数が指定されていません。
undefined identifier	識別子が未定義です。
function argument error：識別子	識別子は関数への引数ではありません。
(missing	(が足りません。
) missing	) が足りません。
} missing	} が足りません。
statement error	構文の誤りです。
case statement error	case 文の誤りです。
default statement error	default 文の誤りです。

4.エラーメッセージ一覧

エラーメッセージ	意　味
switch table overflow	switch 文の case の数があふれました。
more than 1 default	default 文が 2 つ以上あります。
if else statement error	if文とelse文が正しく対応していません。
keyword error：予約語	予約語の用法の誤りです。
undefined struct/union used	未定義の構造体/共用体が使用されています。
struct/untion initalizer over	構造体/共用体の初期化要素が多すぎます。
compound statement error	関数定義後のブロック文の誤りです。
constant expression pequired	定義式でなければいけません。
division by zero	0 で除算しました。
left value required	左辺値でなければいけません。
operand type mismatch	演算数の型が異なります。
type conversion error	型変換が不可能です。
type conversion illegal	型変換が不正です。
thrinary operator error	三項演算子の誤りです。
non-function call	関数以外を呼び出しています。
no left value	アドレス演算子の誤りです。
struct reference error	構造体参照の誤りです。
do not same operand size pointer	演算子の大きさが合っていません。
void function	戻り値無し関数なので戻り値は指定できません。
void type error in expresstion	式の中に値を持たない項があります。
structure declaration error	構造体宣言の誤りです。
argument duplicate definition	引数の宣言とプロトタイプ宣言が重複しています。
illegal use of floating point	浮動小数点値は使用できません。
illegal initiarise error：識別子	識別子は初期化できません。
too few parameters in call	関数呼び出しで引数の個数があっていません。
auto variable 32Kbytes over	自動記憶指定子の変数が 32 キロバイトを超えています。

4.エラーメッセージ一覧

エラーメッセージ	意　　味
structure size 32Kbytes over	構造体変数がcbキロバイトを超えています。
two consecutive dots	関数プロトタイプ宣言で不定個パラメータの宣言が誤りです。
パーザのワーニング	
character constant overflow	文字定数が表現可能な範囲を超えています。
undefined identifier	識別子が未定義です。
void type declarator	void型変数を宣言しています。
type specifier illeagal	型指定子の指定が不正です。
short type specifier illegal	short型指定子の指定が不正です。
unsigned type specifier illegal	unsigned型指定子の指定が不正です。
long type specifier illegal	long型指定子の指定が不正です。
data size 1Mbytes over	データ領域が1メガバイトを超えています。
argument list type mismatch	プロトタイプリストの型と引数数の型が異なっています。 case1： 　　int sub (int, short)； 　　int sub (int, int)； case2： 　　int sub (int, short)； 　　int sub (a, b) 　　register int a； 　　register int b； 　　{ 　　}
function return mismatch	関数の返り値の型があっていません。
struct/union for function argument	構造体/共用体の実体を関数の引数に指定しています。
function return value mismatch	関数の返り値があっていません。

4.エラーメッセージ一覧

エラーメッセージ	意　　味
argument type mismatch	引数の型が異なっています。 case1： 　　int sub (int, short)； 　　int sub (a, b) 　　int a； 　　int b； 　　{ 　　} case2： 　　int sub (int, short)； 　　　　： 　　sub (1. 2f, 0. 0)； 　　　　：
cast conversion	明示的な型変換を行いました。
implied cast conversion	暗黙的な型変換を行いました。
type mismatch	三項演算子の型が異なります。
pointer type mismatch	ポインタの型があっていません。
signed type specifier illegal	signed 型指定子の指定が不正です。
assignment const variable, read-only	const 修飾子の付いた変数に代入しています。読み込み専用です。
size of array is negative	配列の要素数が負の数なので正の数にしました。
identifier length over：識別子	識別子が長すぎます。
label is never used in function	一度も参照されていないラベルが定義されています。
パラメータのエラー	
Illegal identifier max length 文字数	/E スイッチの識別子の文字数指定が不正です。
Illegal warning level レベル数	/A スイッチのワーニングレベル指定が不正です。

4.エラーメッセージ一覧

エラーメッセージ	意　　味
コードジェネレータのエラー	
Duplicate case(16 進値) label(番号)	case 文の数値 (16 進値) が重複しています。
file open error	作業用ファイルのオープンに失敗しました。
Out open error	メモリに空きがありません。
Intermediate file error opcode(コード番号) op(オペレーション)	コードジェネレータの内部エラーです。中間コード又は解析の誤りです。
compiler botch : call	コードジェネレータの内部エラーです。中間コード又は解析の誤りです。
illegal operation on structure	コードジェネレータの内部エラーです。中間コード又は解析の誤りです。
no code table for op : コード番号	コードジェネレータの内部エラーです。中間コード又は解析の誤りです。
division by zero	0で除算しました。
error constand oprand required	定数オペランドが必要な所にありません。
register overflow simplify expression	式が複雑すぎるので作業用レジスタがあふれました。
unimplemented field operator	ビットフィールドにサポートしていない演算を行いました。
Stack overflow botch	コードジェネレータの内部エラーです。中間コード又は解析の誤りです。
Intermediate file error	コードジェネレータの内部エラーです。中間コード又は解析の誤りです。
Expresstion dnput botch	コードジェネレータの内部エラーです。中間コード又は解析の誤りです。
Binary expression botch	コードジェネレータの内部エラーです。中間コード又は解析の誤りです。
Illegal initialization	初期化式の誤りです。

4.エラーメッセージ一覧

エラーメッセージ	意　味
Missing temp file	作業用ファイルがオープンできません。
Can't create ファイル名	作業用ファイルが作成できません。
prins op(コード番号)：(オペレーション)	コードジェネレータの内部エラーです。中間コード又は解析の誤りです。
cprins op(コード番号：(オペレーション)	コードジェネレータの内部エラーです。中間コード又は解析の誤りです。
Illegal use of register	コードジェネレータの内部エラーです。中間コード又は解析の誤りです。
Compiler error：pname op(オペレーション)：(記憶クラスコード番号)	コードジェネレータの内部エラーです。中間コード又は解析の誤りです。
Compiler error pname called illegally op(コード番号)：(オペレーション)	コードジェネレータの内部エラーです。中間コード又は解析の誤りです。
Illegal indirection	コードジェネレータの内部エラーです。中間コード又は解析の誤りです。
pow2 error op(オペレーション)(コード番号)	コードジェネレータの内部エラーです。中間コード又は解析の誤りです。
Uniplementen structure assigment	サポートしていない構造体代入を行いました。
illegal structure operation	サポートしていない構造体の演算を行いました。
パラメータのエラー	
Illegal identifier length	/E スイッチの識別子の文字数指定が不正です。
オプティマイザーのエラー	
soruce file not found(ファイル名)	作業用ファイルがオープンできません。
destination file not found(ファイル名)	作業用ファイルが作成できません。
Sorry, input line too long	テキスト入力用のバッファがあふれました。
Optimizer：Out of sapce	メモリが足りません。

索引 50音順

マ

ヤ

ラ

ワ

アルファベット順

シャープ株式会社
本　　社　〒545 大阪市阿倍野区長池町22番22号
電子機器事業本部　〒329-21 栃木県矢板市早川町174番地
液晶映像システム事業部　第2商品企画部
お問い合わせ先　〒162 東京都新宿区市谷八幡町8番地　電話 (03)3260-1161(大代表)
東京支社内　液晶映像システム事業部　第2商品企画部　ソフトウェア担当